Universidad Nacional de Córdoba

Facultad de Ciencias Exactas Físicas y Naturales

Informe Trabajo Práctico

El rendimiento de las computadoras

Estudiantes:

Cabrera, Augusto Gabriel
Moroz, Esteban Mauricio
Britez, Fabio

Profesores:

Ing Miguel Ángel Solinas
Ing Javier Alejandro Jorge
Ing Silvia Arias

Sistemas de Computación

Córdoba,
22 de marzo de 2024

Índice

1. Enunciado

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.

1.1. Time profiling

En un informe deberán responder a las siguientes preguntas y mostrar con capturas de pantalla la realización del tutorial descripto en time profiling adjuntando las conclusiones sobre el uso del tiempo de las funciones.

1.2. Lista de benchmarks

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 qué benchmark la representa mejor.

- Benchmarks:
  - https://openbenchmarking.org/test/pts/build-linux-kernel-1.15.0
  - https://www.tomshardware.com/reviews/cpu-hierarchy,4312.html

1.3. Rendimiento y aceleración

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

- Intel Core i5-13600K
- AMD Ryzen 9 5900X 12-Core

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

2. Marco Teórico

2.1. Rendimiento

Se define rendimiento de un sistema como la capacidad que tiene dicho sistema para realizar un trabajo en un determinado tiempo. Es inversamente proporcional al tiempo, es decir, cuanto mayor sea el tiempo que necesite, menor será el rendimiento. Los computadores ejecutan las instrucciones que componen los programas, por lo tanto el rendimiento de un computador está relacionado con el tiempo que tarda en ejecutar los programas. De esto se deduce que el tiempo es la medida del rendimiento de un computador.

2.1.1. Comparación de Desempeño

- Comparación en términos relativos (no absolutos)
- Un sistema A se considera que tiene mejor rendimiento que un sistema B si el sistema A tiene un menor tiempo de ejecución (para un conjunto de programas) que el sistema B.

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

Este enfoque de comparación en términos relativos permite evaluar el rendimiento de los sistemas independientemente de las diferencias absolutas en tiempo de ejecución. Por lo tanto, incluso si un sistema tarda más en ejecutar un programa en comparación con otro sistema, puede tener un mejor rendimiento si su tiempo de ejecución es proporcionalmente menor en relación con otros programas o con el mismo programa ejecutado en otro sistema.

Por ejemplo, si el sistema A tarda 20 segundos en ejecutar un programa y el sistema B tarda 30 segundos en ejecutar el mismo programa, se puede concluir que el sistema A tiene un mejor rendimiento. Aunque el sistema A tarda más tiempo que el sistema B, la diferencia relativa entre los tiempos de ejecución sugiere un mejor rendimiento para el sistema A.

2.1.2. Ejemplo

Supongamos que tenemos dos computadoras A y B, y queremos medir su rendimiento en la compilación de un programa.

- La computadora A tarda 10 minutos en compilar el programa.
- La computadora B tarda 5 minutos en compilar el mismo programa.

Podemos ver que la computadora B tiene un mejor rendimiento en comparación con la computadora A, ya que tarda la mitad del tiempo en realizar la misma tarea. Por lo tanto, podemos decir que la computadora B tiene un rendimiento superior en términos de compilación de programas. Esto ilustra la relación inversa entre el tiempo y el rendimiento: cuanto menor sea el tiempo necesario para realizar una tarea, mayor será el rendimiento del sistema.

2.1.3. Rendimiento del procesador

El rendimiento del procesador depende de los siguientes parámetros:

1. Frecuencia de la CPU (f_{CPU}): es el número de ciclos por segundo al que trabaja el procesador o CPU. No confundir la frecuencia de la CPU con la frecuencia del sistema; el bus del sistema trabaja a una frecuencia menor que la CPU.
2. Periodo de la CPU (T_{CPU}): es el tiempo que dura un ciclo y es la inversa de la frecuencia de la CPU.
3. Ciclos por instrucción (CPI): las instrucciones se descomponen en microinstrucciones, que son operaciones básicas que se realizan en un ciclo de reloj. En un programa, el CPI se refiere al promedio de microinstrucciones que tienen las instrucciones del programa, es decir, los ciclos de reloj que se tardan de media en ejecutar una instrucción.
4. Número de instrucciones del programa: cuantas más instrucciones haya en el programa, más tiempo se tarda en ejecutarlo, lo que reduce el rendimiento. El hecho de tener un número reducido de instrucciones depende del programador y de disponer de un buen compilador.
5. Multitarea: hace referencia a la capacidad que tiene un computador para atender simultáneamente varias tareas.

$$f_{\text{CPU}} = \frac{\text{número de ciclos}}{\text{segundos}} \Longrightarrow T_{\text{CPU}} = \frac{1}{f_{\text{CPU}}} \Longrightarrow CPI = \frac{\sum_{i=1}^{n}(N_{\text{instruc}_i} \times CPI_i)}{N_{\text{InstrucTot}}}$$

Se consiguen las siguientes formulaciones:

$$T_{\text{instrucción}} = CPI \times T_{\text{CPU}} \quad ; \quad T_{\text{prog}} = \text{número de instrucciones} \times CPI \times T_{\text{CPU}}$$

$$\eta_{\text{prog}} = \frac{1}{T_{\text{prog}}} = \frac{1}{\text{número de instrucciones} \times CPI \times T_{\text{CPU}}} = \frac{f_{\text{CPU}}}{\text{número de instrucciones} \times CPI}$$

2.1.4. Speedup

Es la razón entre el rendimiento de un sistema mejorado y el rendimiento de su implementación original

$$\text{Speedup} = \frac{\text{Rendimiento mejorado}}{\text{Rendimiento Original}}$$

2.2. Eficiencia

- Mientras que el speedup es la ganancia por mejorar un sistema.
- La eficiencia mide la utilización de un recurso.
- Si Speedup n es la ganancia por mejorar el sistema con n recursos, la eficiencia mide la utilización de esos recursos.

$$\text{Eficiencia} = \frac{\text{Speedup}_n}{n}$$

2.3. Definición de Benchmark

Un benchmark es un conjunto de programas de prueba o programas reales que sirven para medir el rendimiento de un componente concreto o de una computadora en su conjunto, mediante la comparación de los tiempos de ejecución obtenidos de esos programas de prueba con respecto a otras máquinas similares.

2.3.1. Clasificación de Benchmark

1. **Benchmark Sintéticos:** Son programas de prueba que simulan programas reales en carga de trabajo y reparto de instrucciones. Sirven para medir el rendimiento de componentes concretos o de un computador en general. Ejemplos: Whetstone, Dhrystone.

2. **Benchmarks Reducidos:** Consisten en pequeños fragmentos de código con entre 10 y 100 líneas de código que se utilizan para medir una característica concreta del computador. Ejemplos: Java Micro Benchmark, Puzzle, Quicksort, criba de Eratóstenes.

3. **Benchmark Kernel o de Núcleo:** Consiste en un fragmento de código extraído de un programa real. La parte escogida es la más representativa del programa, y por tanto, la parte que más influye en el rendimiento del sistema para ese software.

4. **Programas Reales:** Son los benchmark más utilizados en la actualidad. Consisten en programas reales que son ejecutados con un conjunto de datos reducido para no alargar su ejecución. Ejemplo: benchmark de la familia SPEC, clasificados en SPECint y SPECfp según operen con números enteros o con números reales (en coma flotante).

3. Desarrollo

3.1. Time profiling

Las herramientas para analizar el tiempo de ejecución del programa/uso de memoria se llaman generadores de perfiles. Cómo funcionan los perfiladores de código (tiempo). Los generadores de perfiles de código a menudo se usan para analizar no solo cuánto tiempo tarda en ejecutarse un programa (podemos obtenerlo de herramientas a nivel de shell como /usr/bin/time), sino también cuánto tiempo tarda en ejecutarse cada función o método (tiempo de CPU). Dos técnicas principales utilizadas por los perfiladores: inyección de código, muestreo.

Tenga en cuenta que los bucles 'for' dentro de las funciones están ahí para consumir algo de tiempo de ejecución.

```c
//test_gprof.c
#include<stdio.h>

void new_func1(void);

void func1(void)
{
    printf("\n Inside func1 \n");
    int i = 0;
    for(;i<0xffffffff;i++);
    new_func1();
    return;
}

static void func2(void)
{
    printf("\n Inside func2 \n");
    int i = 0;
    for(;i<0xaffffffff;i++);
    return;
}

int main(void)
{
    printf("\n Inside main()\n");
    int i = 0;
    for(;i<0xfffff11;i++);
    func1();
    func2();
    return 0;
}//test_gprof.c
#include<stdio.h>

void new_func1(void);

void func1(void)
{
```

```
38      printf("\n Inside func1 \n");
39      int i = 0;
40      for(;i<0xffffffff;i++);
41      new_func1();
42      return;
43  }
44
45  static void func2(void)
46  {
47      printf("\n Inside func2 \n");
48      int i = 0;
49      for(;i<0xaffffffff;i++);
50      return;
51  }
52
53  int main(void)
54  {
55      printf("\n Inside main()\n");
56      int i = 0;
57      for(;i<0xfffff11;i++);
58      func1();
59      func2();
60      return 0;
61  }
```

Listing 1: test gprof.c

```
1
2   #include<stdio.h>
3
4   void new_func1(void)
5   {
6       printf("\n Inside new_func1()\n");
7       int i = 0;
8
9       for(;i<0xfffffee;i++);
10
11      return;
12  }
```

Listing 2: test gprof new.c

A continuación se anexan TODOS los comandos ingresados en la bash para poder desarrollar lo solicitado en el pdf **Time profiling**.

```
esteban@esteban-Presario-21-VerK:~$ ls Descargas Escritorio Música Público Documentos Imágenes
Plantillas Vídeos
esteban@esteban-Presario-21-VerK:~$ cd Escritorio
esteban@esteban-Presario-21-VerK:~/Escritorio$ ls SisCom SOII
esteban@esteban-Presario-21-VerK:~/Escritorio$ cd SisCOm
bash: cd: SisCOm: No existe el archivo o el directorio
esteban@esteban-Presario-21-VerK:~/Escritorio$ cd SisCom
esteban@esteban-Presario-21-VerK:~/Escritorio/SisCom$ ls 'Sistemas de Computación.pdf' Tps
esteban@esteban-Presario-21-VerK:~/Escritorio/SisCom$ cd Tps
esteban@esteban-Presario-21-VerK:~/Escritorio/SisCom/Tps$ ls tp1
esteban@esteban-Presario-21-VerK:~/Escritorio/SisCom/Tps$ cd tp1
```

Paso 1: creación de perfiles habilitada durante la compilación

```
esteban@esteban-Presario-21-VerK:~/Escritorio/SisCom/Tps/tp1$ gcc -Wall -pg test_gprof.c
test_gprof_new.c -o test_gprof
```

Paso 2: Ejecutar el código

```
esteban@esteban-Presario-21-VerK:~/Escritorio/SisCom/Tps/tp1$ ls test_gprof test_gprof.c
test_gprof_new.c
esteban@esteban-Presario-21-VerK:~/Escritorio/SisCom/Tps/tp1$ ./test_gprof
Inside main()
Inside func1
Inside new_func1()
Inside func2
esteban@esteban-Presario-21-VerK:~/Escritorio/SisCom/Tps/tp1$ ls gmon.out test_gprof test_gprof.c
test_gprof_new.c
```

Paso 3: Ejecute la herramienta gprof

```
esteban@esteban-Presario-21-VerK:~/Escritorio/SisCom/Tps/tp1$ gprof --version
GNU gprof (GNU Binutils for Ubuntu) 2.38
Based on BSD gprof, copyright 1983 Regents of the University of California.
This program is free software. This program has absolutely no warranty.
esteban@esteban-Presario-21-VerK:~/Escritorio/SisCom/Tps/tp1$ gprof test_gprof gmon.out >
analysis.txt
esteban@esteban-Presario-21-VerK:~/Escritorio/SisCom/Tps/tp1$ ls analysis.txt gmon.out test_gprof
test_gprof.c test_gprof_new.c


esteban@esteban-Presario-21-VerK:~/Escritorio/SisCom/Tps/tp1$ gprof -a test_gprof gmon.out >
analysis2.txt
esteban@esteban-Presario-21-VerK:~/Escritorio/SisCom/Tps/tp1$ gprof -b test_gprof gmon.out >
analysis3.txt
esteban@esteban-Presario-21-VerK:~/Escritorio/SisCom/Tps/tp1$ gprof -p -b test_gprof gmon.out >
analysis4.txt
esteban@esteban-Presario-21-VerK:~/Escritorio/SisCom/Tps/tp1$ gprof -pfunc1 -b test_gprof gmon.out >
analysis5.txt
esteban@esteban-Presario-21-VerK:~/Escritorio/SisCom/Tps/tp1$ gprof -pfunc2 -b test_gprof gmon.out >
analisisFunc2.txt
esteban@esteban-Presario-21-VerK:~/Escritorio/SisCom/Tps/tp1$ gprof -pfunc2 -pfunc1 -b test_gprof
gmon.out > analisisFunc2.txt


esteban@esteban-Presario-21-VerK:~/Escritorio/SisCom/Tps/tp1$ pip instalar gprof2dot
ERROR: unknown command "instalar" - maybe you meant "install"
esteban@esteban-Presario-21-VerK:~/Escritorio/SisCom/Tps/tp1$ pip instalal gprof2dot
ERROR: unknown command "instalal" - maybe you meant "install"
esteban@esteban-Presario-21-VerK:~/Escritorio/SisCom/Tps/tp1$ 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
Successfully installed gprof2dot-2022.7.29
esteban@esteban-Presario-21-VerK:~/Escritorio/SisCom/Tps/tp1$ sudo apt install graphviz
```

3.1.1. Paso 1: Creación de perfiles habilitada durante la compilación

En este primer paso, debemos asegurarnos de 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.

Este comando (-gp) genera código adicional para escribir información de perfil adecuada para el análisis. programa gprof. Debe utilizar esta opción al compilar los archivos fuente que desee. datos sobre, y también debe utilizarlos al vincular.

3.1.2. Paso 2: Ejecutar el código

En el segundo paso, se ejecuta el archivo binario producido como resultado del paso 1 (arriba) para que se pueda generar la información de perfiles.Entonces vemos que cuando se ejecuta el binario, se genera un nuevo archivo 'gmon.out' en el directorio de trabajo actual. Tenga en cuenta que durante la ejecución, si el programa cambia el directorio de trabajo actual (usando chdir), se generará gmon.out en el nuevo directorio de trabajo actual. Además, su programa debe tener permisos suficientes para que gmon.out se cree en el directorio de trabajo actual.

3.1.3. Paso 3: Ejecutar el código

En este paso, la herramienta gprof se ejecuta con el nombre del ejecutable y el 'gmon.out' generado anteriormente como argumento. Esto produce un archivo de análisis que contiene toda la información de perfil deseada.Entonces vemos que se generó un archivo llamado 'analysis.txt'

Comprensión de la información de perfil:

Cómo se produjo anteriormente, toda la información de perfil ahora está presente en *analysis.txt*. Echemos un vistazo a este archivo de texto

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 40.65     12.78    12.78        1    12.78    12.78  new_func1
 34.64     23.67    10.89        1    10.89    23.67  func1
 22.65     30.79     7.12        1     7.12     7.12  func2
  2.07     31.44     0.65                             main

 %         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.

 total     the average number of milliseconds spent in this
ms/call    function and its descendents per call, if this
```

```
            function is profiled, else blank.

name        the name of the function.  This is the minor sort
            for this listing. The index shows the location of
            the function in the gprof listing. If the index is
            in parenthesis it shows where it would appear in
            the gprof listing if it were to be printed.

Call graph (explanation follows)


granularity: each sample hit covers 4 byte(s) for 0.03% of 31.44 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.65   30.79                 main [1]
                10.89   12.78       1/1           func1 [2]
                 7.12    0.00       1/1           func2 [4]
-----------------------------------------------
                10.89   12.78       1/1           main [1]
[2]     75.3   10.89   12.78       1         func1 [2]
                12.78    0.00       1/1           new_func1 [3]
-----------------------------------------------
                12.78    0.00       1/1           func1 [2]
[3]     40.6   12.78    0.00       1         new_func1 [3]
-----------------------------------------------
                 7.12    0.00       1/1           main [1]
[4]     22.6    7.12    0.00       1         func2 [4]
-----------------------------------------------

This table describes the call tree of the program, and was sorted by
the total amount of time spent in each function and its children.

Each entry in this table consists of several lines.  The line with the
index number at the left hand margin lists the current function.
The lines above it list the functions that called this function,
and the lines below it list the functions this one called.
This line lists:
    index   A unique number given to each element of the table.
            Index numbers are sorted numerically.
            The index number is printed next to every function name so
            it is easier to look up where the function is in the table.

    % time  This is the percentage of the 'total' time that was spent
            in this function and its children.  Note that due to
            different viewpoints, functions excluded by options, etc,
            these numbers will NOT add up to 100%.

    self    This is the total amount of time spent in this function.

    children    This is the total amount of time propagated into this
            function by its children.

    called  This is the number of times the function was called.
            If the function called itself recursively, the number
            only includes non-recursive calls, and is followed by
            a '+' and the number of recursive calls.

    name    The name of the current function.  The index number is
            printed after it.  If the function is a member of a
```

```
90              cycle, the cycle number is printed between the
91              function's name and the index number.
92
93  For the function's parents, the fields have the following meanings:
94
95      self    This is the amount of time that was propagated directly
96              from the function into this parent.
97
98      children    This is the amount of time that was propagated from
99              the function's children into this parent.
100
101     called  This is the number of times this parent called the
102             function `/' the total number of times the function
103             was called.  Recursive calls to the function are not
104             included in the number after the `/'.
105
106     name    This is the name of the parent.  The parent's index
107             number is printed after it.  If the parent is a
108             member of a cycle, the cycle number is printed between
109             the name and the index number.
110
111 If the parents of the function cannot be determined, the word
112 `<spontaneous>' is printed in the `name' field, and all the other
113 fields are blank.
114
115 For the function's children, the fields have the following meanings:
116
117     self    This is the amount of time that was propagated directly
118             from the child into the function.
119
120     children    This is the amount of time that was propagated from the
121             child's children to the function.
122
123     called  This is the number of times the function called
124             this child `/' the total number of times the child
125             was called.  Recursive calls by the child are not
126             listed in the number after the `/'.
127
128     name    This is the name of the child.  The child's index
129             number is printed after it.  If the child is a
130             member of a cycle, the cycle number is printed
131             between the name and the index number.
132
133 If there are any cycles (circles) in the call graph, there is an
134 entry for the cycle-as-a-whole.  This entry shows who called the
135 cycle (as parents) and the members of the cycle (as children.)
136 The `+' recursive calls entry shows the number of function calls that
137 were internal to the cycle, and the calls entry for each member shows,
138 for that member, how many times it was called from other members of
139 the cycle.
140
141 Index by function name
142
143    [2] func1                   [1] main
144    [4] func2                   [3] new_func1
145
146
```

La información proporcionada te ofrece una visión detallada del rendimiento y la estructura de llamadas de tu programa. Esto puede ser útil para identificar qué partes del código consumen más

tiempo de ejecución, dónde se pueden hacer mejoras de rendimiento y comprender cómo interactúan las diferentes funciones en tu programa. Además, te permite entender la jerarquía de llamadas entre funciones y cómo se propagan los tiempos de ejecución a través de ellas. En resumen, esta información te ayuda a optimizar y comprender mejor el comportamiento de tu programa.

Suprima la impresión de funciones declaradas estáticamente (privadas) usando -a

A continuación, se visualiza el archvio *analysis2.txt.***(NOTA: Entonces vemos que no hay información relacionada con func2 (que se define como estática))**

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 57.28     18.01    18.01        2     9.01    15.39  func1
 40.65     30.79    12.78        1    12.78    12.78  new_func1
  2.07     31.44     0.65                             main

 %         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.

 total     the average number of milliseconds spent in this
ms/call    function and its descendents per call, if this
           function is profiled, else blank.

name       the name of the function.  This is the minor sort
           for this listing. The index shows the location of
           the function in the gprof listing. If the index is
           in parenthesis it shows where it would appear in
           the gprof listing if it were to be printed.
```

```
Copyright (C) 2012-2022 Free Software Foundation, Inc.

Copying and distribution of this file, with or without modification,
are permitted in any medium without royalty provided the copyright
notice and this notice are preserved.
```

```
         Call graph (explanation follows)


granularity: each sample hit covers 4 byte(s) for 0.03% of 31.44 seconds

index % time    self  children    called     name
```

```
                                                <spontaneous>
[1]    100.0    0.65   30.79                 main [1]
               18.01   12.78       2/2           func1 [2]
-----------------------------------------------
               18.01   12.78       2/2           main [1]
[2]     97.9   18.01   12.78       2         func1 [2]
               12.78    0.00       1/1           new_func1 [3]
-----------------------------------------------
               12.78    0.00       1/1           func1 [2]
[3]     40.6   12.78    0.00       1         new_func1 [3]
-----------------------------------------------

This table describes the call tree of the program, and was sorted by
the total amount of time spent in each function and its children.

Each entry in this table consists of several lines.  The line with the
index number at the left hand margin lists the current function.
The lines above it list the functions that called this function,
and the lines below it list the functions this one called.
This line lists:
    index A unique number given to each element of the table.
    Index numbers are sorted numerically.
    The index number is printed next to every function name so
    it is easier to look up where the function is in the table.

     % time This is the percentage of the `total' time that was spent
    in this function and its children.  Note that due to
    different viewpoints, functions excluded by options, etc,
    these numbers will NOT add up to 100%.

     self This is the total amount of time spent in this function.

     children This is the total amount of time propagated into this
    function by its children.

     called This is the number of times the function was called.
    If the function called itself recursively, the number
    only includes non-recursive calls, and is followed by
    a `+' and the number of recursive calls.

     name The name of the current function.  The index number is
    printed after it.  If the function is a member of a
    cycle, the cycle number is printed between the
    function's name and the index number.


 For the function's parents, the fields have the following meanings:

     self This is the amount of time that was propagated directly
    from the function into this parent.

     children This is the amount of time that was propagated from
    the function's children into this parent.

     called This is the number of times this parent called the
    function `/' the total number of times the function
    was called.  Recursive calls to the function are not
    included in the number after the `/'.

     name This is the name of the parent.  The parent's index
```

```
    number is printed after it.  If the parent is a
    member of a cycle, the cycle number is printed between
    the name and the index number.

 If the parents of the function cannot be determined, the word
 '<spontaneous>' is printed in the 'name' field, and all the other
 fields are blank.

 For the function's children, the fields have the following meanings:

     self This is the amount of time that was propagated directly
    from the child into the function.

     children This is the amount of time that was propagated from the
    child's children to the function.

     called This is the number of times the function called
    this child '/' the total number of times the child
    was called.  Recursive calls by the child are not
    listed in the number after the '/'.

     name This is the name of the child.  The child's index
    number is printed after it.  If the child is a
    member of a cycle, the cycle number is printed
    between the name and the index number.

 If there are any cycles (circles) in the call graph, there is an
 entry for the cycle-as-a-whole.  This entry shows who called the
 cycle (as parents) and the members of the cycle (as children.)
 The '+' recursive calls entry shows the number of function calls that
 were internal to the cycle, and the calls entry for each member shows,
 for that member, how many times it was called from other members of
 the cycle.

Copyright (C) 2012-2022 Free Software Foundation, Inc.

Copying and distribution of this file, with or without modification,
are permitted in any medium without royalty provided the copyright
notice and this notice are preserved.

Index by function name

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

El segundo conjunto de datos no menciona la función (func2), que estaba presente en el primer conjunto de datos. Esto sugiere que la función (func2) no se ha ejecutado o no ha sido perfilada en esta instancia del programa. La ausencia de esta función en el segundo conjunto de datos indica un cambio en la ejecución o en la forma en que se ha recopilado la información de perfilado.

Elimine los textos detallados usando -b

Se observa a continuación el archivo *analysis3.txt*

```
Flat profile:
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 40.65     12.78    12.78        1    12.78    12.78  new_func1
```

```
 34.64      23.67     10.89         1     10.89     23.67  func1
 22.65      30.79      7.12         1      7.12      7.12  func2
  2.07      31.44      0.65                               main

granularity: each sample hit covers 4 byte(s) for 0.03% of 31.44 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.65   30.79                 main [1]
               10.89   12.78       1/1           func1 [2]
                7.12    0.00       1/1           func2 [4]
-----------------------------------------------
               10.89   12.78       1/1           main [1]
[2]     75.3   10.89   12.78       1         func1 [2]
               12.78    0.00       1/1           new_func1 [3]
-----------------------------------------------
               12.78    0.00       1/1           func1 [2]
[3]     40.6   12.78    0.00       1         new_func1 [3]
-----------------------------------------------
                7.12    0.00       1/1           main [1]
[4]     22.6    7.12    0.00       1         func2 [4]
-----------------------------------------------
Index by Function Name
   [2] func1                   [1] main
   [4] func2                   [3] new_func1
```

Imprima solo perfil plano usando -p

Se observa a continuación el archivo *analysis4.txt*

```
Flat profile:
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 40.65     12.78    12.78        1    12.78    12.78  new_func1
 34.64     23.67    10.89        1    10.89    23.67  func1
 22.65     30.79     7.12        1     7.12     7.12  func2
  2.07     31.44     0.65                             main
```

Imprimir información relacionada con funciones específicas en perfil plano

Se observa a continuación el archivo *analysis5.txt*

```
Flat profile:
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
100.00     10.89    10.89        1    10.89    10.89  func1
```

Tambien se observa que se pueden incorporar varias funciones ademas de una sola.

Genere un gráfico gprof2dot es una herramienta que puede crear una visualización de la salida de gprof

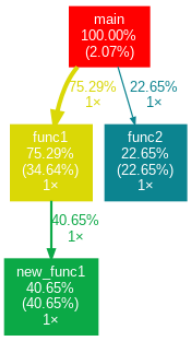

Figura 1: Salida de gprof

Perfilado con linux perf

Perf es una pequeña herramienta que acabo de encontrar para crear perfiles de programas. Perf utiliza perfiles estadísticos, donde sondea el programa y ve qué función está funcionando. Esto es menos preciso, pero tiene menos impacto en el rendimiento que algo como Callgrind, que rastrea cada llamada. Los resultados siguen siendo razonablemente precisos, e incluso con menos muestras, mostrará qué funciones están tomando mucho tiempo, incluso si pierde funciones que son muy rápidas (que probablemente no sean las que está buscando al perfilar de todos modos).

Para la ejecución de perf, necesitaba primero conocer la versión de Kernel en el dispositivo en cuestión, para ello, se ejecutó la siguiente secuencia de comandos en la bash.

```
augusto@augustoCabrera:~$ uname -r
```

Se obtiene la siguiente salida, que representa mi version de kernel:

```
augusto@augustoCabrera:~$ uname -r
6.5.0-26-generic
```

y ahora, ejecutamos el siguiente comando con esa versión de kernel:

```
augusto@augustoCabrera:~$ sudo apt install linux-tools-6.5.0-26-generic
```

y continuamos con los siguientes comandos:

```
augusto@augustoCabrera:~/Escritorio/Facultad/Sistemas de computacion/examples$ chmod
    ↪ +x test_gprof
augusto@augustoCabrera:~/Escritorio/Facultad/Sistemas de computacion/examples$ cd ..
augusto@augustoCabrera:~/Escritorio/Facultad/Sistemas de computacion$ sudo perf
    ↪ record ./examples/test_gprof

 Inside main()

 Inside func1

 Inside new_func1()

 Inside func2
[ perf record: Woken up 13 times to write data ]
[ perf record: Captured and wrote 3,233 MB perf.data (84115 samples) ]
augusto@augustoCabrera:~/Escritorio/Facultad/Sistemas de computacion$ sudo perf
    ↪ report
augusto@augustoCabrera:~/Escritorio/Facultad/Sistemas de computacion$ sudo perf
    ↪ report
```

```
Samples: 84K of event 'cycles:P', Event count (approx.): 84214752808
Overhead  Command     Shared Object      Symbol
  36,45%  test_gprof  test_gprof         [.] func1
  36,18%  test_gprof  test_gprof         [.] new_func1
  24,83%  test_gprof  test_gprof         [.] func2
   2,23%  test_gprof  test_gprof         [.] main
   0,02%  test_gprof  [kernel.kallsyms]  [k] read_hpet
   0,02%  test_gprof  [kernel.kallsyms]  [k] __raw_spin_lock_irqsave
   0,02%  test_gprof  [kernel.kallsyms]  [k] _raw_spin_unlock_irqrestore
   0,02%  test_gprof  [kernel.kallsyms]  [k] handle_tx_event
   0,01%  test_gprof  [kernel.kallsyms]  [k] uvc_video_decode_isoc
   0,01%  test_gprof  [kernel.kallsyms]  [k] srso_return_thunk
   0,01%  test_gprof  [kernel.kallsyms]  [k] fast_mix
   0,01%  test_gprof  [kernel.kallsyms]  [k] xhci_handle_event
   0,01%  test_gprof  [kernel.kallsyms]  [k] srso_safe_ret
   0,01%  test_gprof  [kernel.kallsyms]  [k] finish_td.isra.0
   0,01%  test_gprof  [kernel.kallsyms]  [k] uvc_video_stats_decode
   0,00%  test_gprof  [kernel.kallsyms]  [k] xhci_td_cleanup.isra.0
   0,00%  test_gprof  [kernel.kallsyms]  [k] process_isoc_td.isra.0
   0,00%  test_gprof  [kernel.kallsyms]  [k] uvc_video_decode_start
   0,00%  test_gprof  [kernel.kallsyms]  [k] __queue_work
   0,00%  test_gprof  [kernel.kallsyms]  [k] apic_ack_irq
   0,00%  test_gprof  [kernel.kallsyms]  [k] run_posix_cpu_timers
   0,00%  test_gprof  [kernel.kallsyms]  [k] usb_hcd_giveback_urb
   0,00%  test_gprof  [kernel.kallsyms]  [k] perf_adjust_freq_unthr_context
   0,00%  test_gprof  [kernel.kallsyms]  [k] __get_user_8
   0,00%  test_gprof  [kernel.kallsyms]  [k] tick_sched_handle
   0,00%  test_gprof  [kernel.kallsyms]  [k] nohz_balancer_kick
   0,00%  test_gprof  [kernel.kallsyms]  [k] complete_signal
Cannot load tips.txt file, please install perf!
```

Figura 2: sudo perf report

La tabla que se muestra al ejecutar el comando perf report ordena los datos en varias columnas:

- La columna Gastos generales: Indica qué porcentaje de las muestras globales se recogieron en esa función concreta.
- La columna Comando: Indica en qué proceso se recogieron las muestras.
- La columna Objeto compartido: Muestra el nombre de la imagen ELF de la que provienen las muestras (el nombre [kernel.kallsyms] se utiliza cuando las muestras provienen del kernel).
- La columna Símbolo: Muestra el nombre o símbolo de la función. En el modo por defecto, las funciones se clasifican en orden descendente, mostrando primero las que tienen mayor sobrecarga.

3.2. Preguntas

3.2.1. Benchmark

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.

Tareas Diarias	**Benchmark Representativo**	**Explicación**
Navegar por Internet	Benchmark de Navegación Web (Real)	Los benchmarks reales simulan el comportamiento de un navegador web típico.
Enviar y recibir correos electrónicos	Benchmark de Correo Electrónico (Real)	Los benchmarks reales simulan el tráfico de correo electrónico y la gestión de bandejas de entrada.
Reproducir videos en línea	Benchmark de Transmisión de Video (Real)	Evalúa la capacidad del sistema para manejar la reproducción de videos en tiempo real.
Edición de documentos de texto	Benchmark de Procesamiento de Texto (Kernel)	El benchmark kernel evalúa la velocidad de procesamiento de tareas específicas de edición de texto.
Reproducir música en línea	Benchmark de Transmisión de Audio (Real)	Mide la capacidad del sistema para transmitir audio sin problemas.
Realizar cálculos matemáticos	Benchmark Sintético de Cálculos (Sintético)	Los benchmarks sintéticos son adecuados para medir la potencia de cálculo pura de la CPU.
Ejecutar aplicaciones de productividad	Benchmark de Productividad (Real)	Los benchmarks reales simulan el uso típico de aplicaciones de productividad, como suites de oficina.
Jugar videojuegos	Benchmark de Rendimiento de Juegos (Real)	Los benchmarks reales proporcionan una representación precisa del rendimiento del sistema en escenarios de juego.

Cuadro 1: Tabla de Tareas Diarias con Benchmark Representativo

A continuación, se observan benchmark en el comercio actual:

- Lectura/Escritura de archivos en disco ⟵ CrystalDiskMark
- Velocidad de internet ⟵ Speedtest by Ookla / Fast by Netflix
- Ejecución de programas ⟵ SPEC CPU
- Rendimiento gráfico ⟵ 3DMark
- Memoria ram ⟵ MemTest86

3.2.2. Calculo de rendimientos entre procesadores

¿Cual es el rendimiento de estos procesadores para compilar el kernel de linux ? Intel Core i5-13600K y AMD Ryzen 9 5900X 12-Core ¿Cual es la aceleración cuando usamos un AMD Ryzen 9 7950X 16-Core

El gráfico a continuación, denota la diferencia entre **los tiempos de Compilación de los kernel de LINUX entre los dos procesadores**

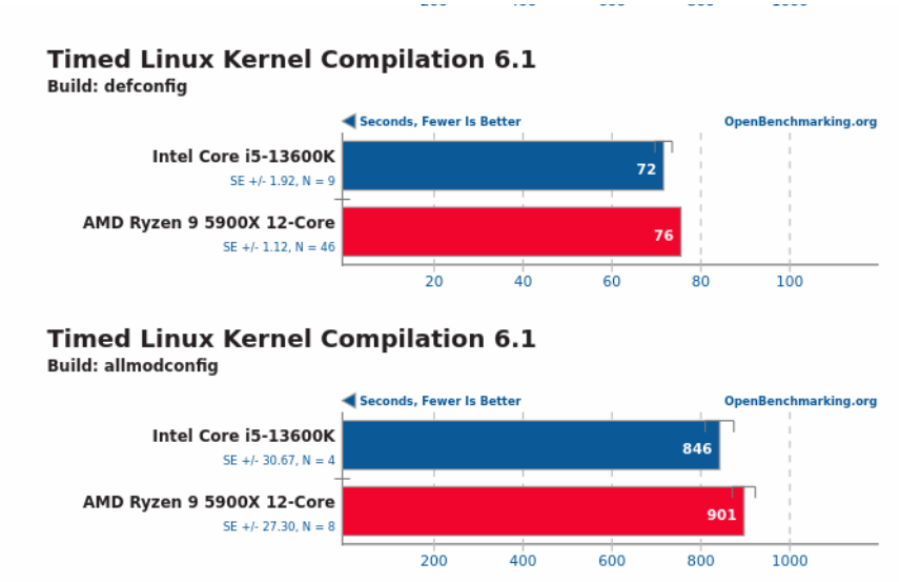

Figura 3: Tiempos de Compilación de los kernel de LINUX

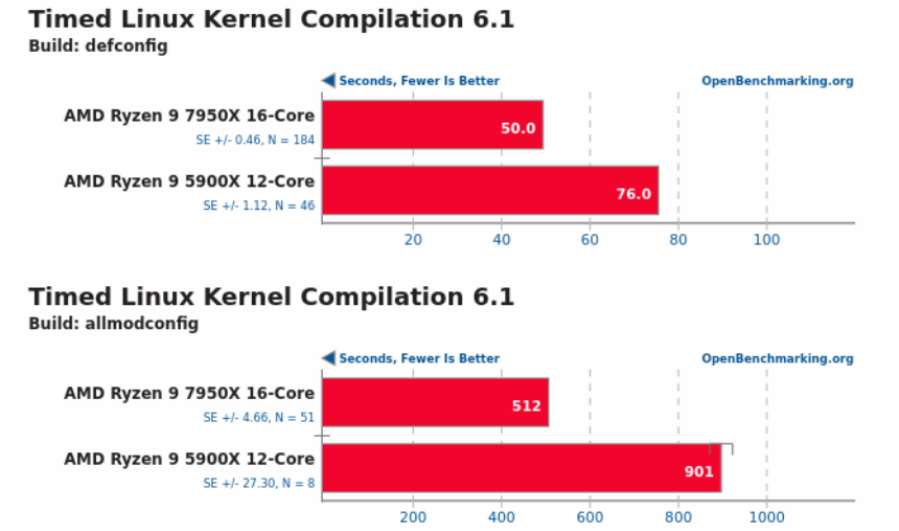

Figura 4: Tiempos de Compilación de los kernel de LINUX entre AMD RYZEN 9 7950X 16-CORE Y 5900X 12-CORE

Se observa el rendimiento de:

Procesador	***N° NUCLEOS***	***TIEMPO [s]***	***RENDIMIENTO***
AMD RYZEN 9 5900X 12-CORE	12	76	$1/76 = 0.013157$
INTEL CORE i5 13600K	14	72	$1/72 = 0.013888$
AMD RYZEN 9 7950X 16-CORE	16	50	$1/50 = 0.02$

$$\text{Speedup} = \frac{\text{Rendimiento mejorado}}{\text{Rendimiento Original}} = \frac{\text{AMD RYZEN 9 7950X 16-CORE}}{\text{AMD RYZEN 9 5900X 12-CORE}} = \frac{0{,}02}{0{,}013157} = 1,52$$

En términos de mejora, obtuvimos un 52 % adicional. Por ende, a pesar de la diferencia económica, es tentadora la opción del cambio de procesador. Adicionalmente, el **INTEL CORE i5 13600K** muestra una pequeña mejora en terminos de rendimiento.

$$\begin{aligned}\text{Speedup} &= \frac{To}{Tm} \\ \frac{72}{72} = 1 \quad &\longleftrightarrow \quad \text{Eficiencia} = \frac{1}{12} = 0{,}08333 \\ \frac{76}{72} = 1{,}055 \quad &\longleftrightarrow \quad \text{Eficiencia} = \frac{1{,}055}{14} = 0{,}07539 \\ \frac{76}{50} = 1{,}52 \quad &\longleftrightarrow \quad \text{Eficiencia} = \frac{1{,}52}{16} = 0{,}095\end{aligned}$$

Con esto, se comprueba que el mas eficiente es el **AMD RYZEN 9 7950X 16-CORE**. Ahora, considerando los precios en la plataforma de *Amazon*.

- AMD RYZEN 9 5900X 12-CORE cuesta 260 dolares.
- INTEL CORE i5 13600K cuesta 290 dolares.
- AMD RYZEN 9 7950X 16-CORE cuesta 550 dolares.

$$\begin{aligned}\text{Speedup} &= \frac{To}{Tm} \\ \frac{72}{72} = 1 \quad &\longleftrightarrow \quad \text{Eficiencia} = \frac{1}{260} = 0,003846 \\ \frac{76}{72} = 1{,}055 \quad &\longleftrightarrow \quad \text{Eficiencia} = \frac{1{,}055}{290} = 0,0036337 \\ \frac{76}{50} = 1{,}52 \quad &\longleftrightarrow \quad \text{Eficiencia} = \frac{1{,}52}{550} = 0,002763\end{aligned}$$

Como conclusión, a la relación eficiencia/precio se observa que **AMD RYZEN 9 5900X 12-CORE** es el mejor.