Universidad de Alcalá
Escuela Politécnica Superior

Grado en Ingeniería de Computadores

Trabajo Fin de Grado

Estudio e implementación de estrategias para el aislamiento de tareas de tiempo real en sistemas multinúcleo

ESCUELA POLITECNICA SUPERIOR

Autor: Ángel Alonso Sánchez

Tutor: Antonio da Silva Fariña

2022

UNIVERSIDAD DE ALCALÁ

ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería de Computadores

Trabajo Fin de Grado

Estudio e implementación de estrategias para el aislamiento de tareas de tiempo real en sistemas multinúcleo

Autor: Ángel Alonso Sánchez

Tutor: Antonio da Silva Fariña

Tribunal:

Presidente: Óscar Rodríguez Polo

Vocal 1º: Pablo Parra Espada

Vocal 2º: Antonio da Silva Fariña

Fecha de depósito: Junio de 2022

Por el apoyo recibido de mi familia, amigos y sobro todo tuyo, Laura.

“0 %2 %. %3 %#3/ %3$)& %2 %4̇ %).#)42,$ %3 2$ %.
0/215 %#5!.$/,!4)2!.)! %3, %9,!52 %6/,5#)/. %3/2$ %.”
René.

Resumen

Hoy en día un sistema de sobremesa de propósito general equipado con un procesador i7, común hoy en día, dispone de varios núcleos que proporcionan una gran potencia de procesamiento software. Este recurso es gestionado por los sistemas operativos de propósito general como Windows o Linux, balanceando la ejecución de procesos entre los diferentes núcleos e incluso migrando los procesos de un núcleo a otro si el planificador lo estima oportuno.

En la mayoría de los casos esta gestión es la adecuada. Sin embargo, existen situaciones donde el uso de un núcleo en exclusiva mejora la respuesta temporal de la aplicación. Por ello, la posibilidad de aislar núcleos de un procesador es una aproximación interesante a la hora de ejecutar ciertas aplicaciones que necesitan ser ejecutadas en tiempo real, con requisitos de tiempo de respuesta muy cortos.

Si se ejecutan estas aplicaciones en varios núcleos de forma simultánea como se suelen ejecutar el resto de las aplicaciones, la carga del planificador más las interrupciones se reparten entre los distintos núcleos, puede generar retardos en la aplicación que hacen impredecible la respuesta temporal de la misma.

Estos retardos no son significativos a la hora de usar aplicaciones o lanzar tareas a nivel de usuario convencional, pero, sin embargo, cuando se tratan de aplicaciones en tiempo real, estos pequeños retardos pueden afectar al correcto funcionamiento de estas aplicaciones ya que requieren tiempos de respuesta específicos.

A lo largo de este trabajo de fin de grado se van a analizar distintos mecanismos para el aislamiento de un núcleo, explicando en que consiste cada mecanismo y como se pueden utilizar, realizando pruebas con cada uno de estos tipos de mecanismos y observando los resultados que se obtienen de estas pruebas. De esta forma se podrá determinar cuál de estos mecanismos podría ser más efectivo para reducir los tiempos de respuesta y la latencia en las operaciones y, por tanto, ofrecer un mejor aislamiento del núcleo de ejecución.

Abstract

Nowadays a general purpose desktop system equipped with an i7 processor, which is common today, has several cores that provide great software processing power. This resource is managed by general purpose operating systems such as Windows or Linux, balancing the execution of processes between the different cores and even migrating processes from one core to another if the scheduler deems it appropriate.

In most cases this management is adequate. However, there are situations where the exclusive use of one core improves the application's time response. Therefore, the possibility of isolating cores of a processor is an interesting approach when running certain applications that need to be executed in real time, with very short response time requirements.

If these applications are executed on several cores simultaneously in the same way as other applications are usually executed, the load of the scheduler plus the interrupts are distributed among the different cores, can generate delays in the application that make its time response unpredictable.

These delays are not significant when using applications or launching tasks at the conventional user level, but nevertheless, when dealing with real-time applications, these small delays can affect the correct operation of these applications as they require limited response times.

Throughout this thesis we will analyze different mechanisms for the isolation of a kernel, explaining what each mechanism consists of and how they can be used, performing tests with each of these types of mechanisms and observing the results obtained from these tests. In this way it will be possible to determine which of these mechanisms could be more effective to reduce response times and latency in operations and, therefore, provide a better isolation of the execution core.

Índice general

Resumen **vii**

Abstract **ix**

Índice general **xi**

Índice de figuras **xiii**

Índice de tablas **xv**

1 Introducción **1**

1.1 Presentación 1

1.2 Sistemas en Tiempo Real 2

1.3 Linux para Tiempo Real 2

1.4 Aislamiento de Núcleos para tareas de Tiempo Real 3

1.5 Organización de la memoria 3

2 Estudio teórico **5**

2.1 Introducción 5

2.2 Marco tecnológico 5

2.3 Aislamiento de la CPU 9

2.3.1 Isolcpus 9

2.3.2 CPUSET 9

2.4 Configuración y verificación de CPUSET 9

2.5 Afinidad de las interrupciones 11

2.6 Afinidad de los procesos 12

2.7 Aislamiento de núcleos en Windows 12

3 Desarrollo **15**

3.1 Introducción 15

3.2 Desarrollo del sistema de experimentación 15

3.3 Latencygraph.sh 16

3.3.1 Funcionamiento del script 16
3.3.2 Código del script 17
3.3.2.1 Variables 17
3.3.2.2 Funciones 17
3.3.2.3 Main 18
3.4 Raspberry 21
3.4.1 enable_ccr.c 22
3.4.2 signal.c 25

4 Resultados 31
4.1 Introducción 31
4.2 Entorno experimental 31
4.2.1 Ubuntu 31
4.2.1.1 Hackbench 32
4.2.1.2 Pruebas con Taskset 32
4.2.1.3 Pruebas con CPUSET 34
4.2.1.4 Stress 36
4.2.1.5 Pruebas con Taskset 37
4.2.1.6 Pruebas con CPUSET 39
4.2.1.7 Sysbench 41
4.2.1.8 Pruebas con Taskset 42
4.2.1.9 Pruebas con CPUSET 44
4.2.2 Pruebas con la Raspberry 46
4.2.2.1 Ejecución nominal 47
4.2.2.2 Ejecución mediante Taskset en un CPUSET aislado 48

5 Conclusiones y líneas futuras 51

Bibliografía 53

Apéndice A Código fuente Latencygraph 55

Apéndice B Códigos fuentes usados en la Raspberry 59
B.1 enable_ccr.c 59
B.2 signal.c 60
B.3 Makefile 62

Apéndice C Herramientas y recursos 63

Índice de figuras

2.1 Ejecución de un programa en Linux con cuatro núcleos. 6

2.2 Análisis de la ejecución de un programa en Linux con cuatro núcleos. 7

2.3 Objetivo de ejecución de una aplicación en un núcleo aislado. 8

2.4 Análisis del objetivo del aislamiento de un núcleo. 8

2.5 Creación y montaje de cpuset. 10

2.6 Creación de carpeta nueva dentro de cpuset. 10

2.7 Configuración de los parámetros de cpuset. 10

2.8 Verificación de la creación del cpuset llamado *mycpuset* 11

2.9 Ejemplo de utilización y configuración de la máscara de bits para determinar qué núcleos del procesador atenderán una interrupción. 11

3.1 Diagrama de flujo del script. 16

3.2 Variables del script. 17

3.3 Funciones del script. 18

3.4 Switch case para la elección del benchmark a ejecutar. 19

3.5 Bloque de comandos para la obtención de los datos de Cyclictest. 20

3.6 Título y nombres de los ejes de la gráfica. 20

3.7 Anexión de los datos para la leyenda. 21

3.8 Generación de gráfica y eliminación de archivos temporales. 21

3.9 Función enable_ccr(void *info). 22

3.10 Diagrama de nivel superior del procesador Cortex-A7 MPCore. 23

3.11 Bits de asignación para PMUSERENR [1, Capítulo D13.5.17]. 24

3.12 Asignación de bits para el PMCR. 24

3.13 Asignación de bits para el PMCNTENSET. 25

3.14 Función init_module(void). 25

3.15 Preparación del entorno de la Raspberry. 26

3.16 Definición y asignación de las variables time_high y time_low. 26

3.17 Preparación de GPIO. 27

3.18 Función mmap_bcm_register(). 27

3.19 Función initialize_gpio_for_output(). 28
3.20 Detalle del registro de configuración GPSEL2. 28
3.21 Bucle para la lectura de los valores del contador. 29
3.22 Función para la lectura del PMCCNTR. 29

4.1 Hackbench simple. 32
4.2 Hackbench con T simple. 33
4.3 Hackbench con T juntos. 33
4.4 Hackbench con T separados. 34
4.5 Hackbench on CPU2 and graphic on all CPUs. 35
4.6 Hackbench and graphic on CPU2. 35
4.7 Hackbench on CPU2y gráficos en CPU3. 36
4.8 Stress simple. 37
4.9 Stress con T simple. 38
4.10 Stress con T juntos. 38
4.11 Stress con T separados. 39
4.12 Stress en CPU2 y gráficos en todas las CPUs. 40
4.13 Stress y gráficos en CPU2. 40
4.14 Stress en CPU2 y gráficos en CPU3. 41
4.15 Sysbench simple. 42
4.16 Sysbench con T simple. 43
4.17 Sysbench con T juntos. 43
4.18 Sysbench con T separados. 44
4.19 Sysbench en CPU2 y gráficos en todas las CPUs. 45
4.20 Sysbench y gráficos en CPU2. 45
4.21 Sysbench en CPU2 y gráficos en CPU3. 46
4.22 Imagen obtenida del osciloscopio con un Time de 50 ns. 46
4.23 Datos de la ejecución nominal en Raspberry. 47
4.24 Creación de los CPUSET. 48
4.25 Definición de la afinidad de las interrupciones. 48
4.26 Definición de los eventos de planificación y ejecución del programa de prueba. 49
4.27 Datos de la ejecución con Taskset en Raspberry. 49

Índice de tablas

2.1 Tabla de referencia para escoger las CPUs. 13

3.1 Habilitar registro de usuario. 23

3.2 Habilitación del PMCR. 24

3.3 Habilitar el recuento de los monitores de rendimiento. 24

3.4 Acceso al PMCCNTR. 29

4.1 Media y desviación típica nominal 48

4.2 Media y desviación típica usando taskset 49

Capítulo 1

Introducción

[...] Éste es el secreto de la felicidad y la virtud: amar lo que uno tiene que hacer.

Aldous Huxley, Un mundo feliz.

1.1 Presentación

El software se ha convertido en el elemento diferenciador de muchos productos, mientras que los aspectos mecánicos y el hardware (electrónica) se están convirtiendo en tecnologías de soporte. Además, la arquitectura de los sistemas tiende a separar cada vez más los aspectos hardware del software para permitir dos procesos de desarrollo en gran medida independientes. Así, el software puede actualizarse con frecuencia, incluso antes de que el producto esté finalizado como después de que se haya puesto en producción. Como parte de esta tendencia, los clientes esperan cada vez más que el software de sus productos evolucione proporcionando nuevas funcionalidades e incluso solucionando problemas o mal funcionamientos que puedan surgir durante el tiempo de vida útil del sistema. Además, al ser el software un elemento "blando" que es el más fácil de moldear y cambiar, es, al mismo tiempo, la forma más sencilla de hacer concesiones e incorporar comportamientos no deseados en el producto final.

Los sistemas multinúcleo se vienen usando desde hace tiempo en el ámbito de los ordenadores de sobremesa y la supercomputación. Sin embargo, su uso no había sido habitual en el dominio de aplicación de los sistemas empotrados [2]. Pero actualmente la complejidad de los diseños y el aumento del software parece ser hoy en día una tendencia imparable. De acuerdo con la organización ITRS [3], podría haber 6000 procesadores integrados en un único encapsulado para 2026. Diseñar y verificar el software empotrado en sistemas multinúcleo representa un enorme reto.

Actualmente un sistema de sobremesa de propósito general equipado con un procesador i7, común hoy en día, dispone de varios núcleos que proporcionan una gran potencia de procesamiento software. Este recurso es gestionado por los sistemas operativos de propósito general como Windows o Linux, balanceando la ejecución de procesos entre los diferentes núcleos e incluso migrando los procesos de un núcleo a otro si el planificador lo estima oportuno.

1.2 Sistemas en Tiempo Real

El término “Sistema en Tiempo Real”, hace referencia a todos aquellos sistemas informáticos que se encuentran en constante interacción con el entorno y, además, tienen que responder a los eventos dentro de unas limitaciones de tiempo predecibles y específicas. Si esto último no ocurriera, las consecuencias podrían ser desastrosas [4].

Esta clase de sistemas está presente en muchos aspectos de nuestro día a día. Los podemos hallar en decenas de automatismos de un coche (airbag, ABS, inyección, ...), en la distribución de la energía eléctrica, en sistemas de fabricación integrada, en el control de edificios, en telefonía móvil o en sistemas multimedia y en aviónica y/o sistemas embarcados para aplicaciones espaciales. Su uso está tan extendido que actualmente el número de estos sistemas en funcionamiento es superior al de sistemas computacionales de propósito general.

No todos los sistemas de tiempo real tienen el mismo nivel de restricción o criticidad. Por lo tanto, estos se pueden clasificar en función de la criticidad asociada a los plazos temporales de respuesta:

- **Tiempo real estricto (hard real-time)**: En esta primera clasificación, todas las respuestas que debe dar el sistema tienen que terminar dentro de un plazo temporal especificado. Si el plazo de tiempo para dar una respuesta o realizar una acción concreta no se cumple puede haber consecuencias catastróficas en el sistema. Un ejemplo de esto puede ser el accionamiento del airbag del coche, en el que si tiene una respuesta superior a la permitida puede ocasiones graves consecuencias.
- **Tiempo real blando (soft real-time)**: Esta vez, los tiempos de respuesta son importantes pero el sistema puede seguir funcionando, aunque algún plazo temporal no se cumpla. Normalmente en esta clase de sistemas, el incumplimiento de los plazos temporales produce un empeoramiento del rendimiento. Un ejemplo de un sistema de tiempo real blando es el de reproducir una película en streaming ya que, aunque se pudiese parar, no implicaría que no se pueda seguir viendo cuando reanude o vaya a producirse una situación catastrófica.

En los sistemas informáticos un sistema operativo es el software encargado de gestionar los recursos hardware y ofrecer los servicios que necesitan las aplicaciones. Dentro de los sistemas operativos podemos encontrar aquellos que son de tiempo real, que han sido diseñados para no solo ofrecer corrección lógica en los resultados computados, sino para ofrecer garantías temporales en la ejecución de las aplicaciones que gestionan.

1.3 Linux para Tiempo Real

Los sistemas operativos basados en el kernel de Linux, a pesar de tener un claro propósito de uso general, han ido evolucionando a lo largo del tiempo en términos de características y comportamiento temporal, y por ende han ampliado su capacidad para ser utilizados en entornos más específicos. Prueba de ello es el esfuerzo realizado por diversos desarrolladores para mejorar su rendimiento en entornos donde el comportamiento temporal de las aplicaciones es un requisito.

Mediante de la modificación de aspectos claves del comportamiento interno del kernel del sistema con el parche denominado PREEMPT_RT [5], se ha conseguido mejorar considerablemente sus características para utilizarse en entornos con demandas temporales. De este modo, utilizando el parche PREEMPT_RT se consigue un mayor grado de predictibilidad en los tiempos de respuesta de los servicios proporcionados por el sistema operativo. Además, cabe destacar que gran parte de las características de tiempo real ofrecidas por el parche PREEMPT_RT se han ido añadiendo a la línea principal del kernel de Linux.

Desde la versión 2.6 del kernel de Linux tenemos la opción denominada CONFIG_PREEMPT que si es activada antes de la compilación del kernel hace que gran parte del código sea expulsable [6]). Aunque el kernel seguirá siendo no expulsable cuando las interrupciones estén deshabilitadas. Además, seguían quedando aun así partes del código no expulsables en la versión principal del kernel de Linux, lo que provoca que en determinadas situaciones no tengamos tiempos de respuesta acotados para las tareas de tiempo real. Por este motivo uno de los principales objetivos del parche PREEMPT_RT es intentar reducir al máximo las secciones del código que no son expulsables.

1.4 Aislamiento de Núcleos para tareas de Tiempo Real

Como se ha comentado en el apartado anterior, existen implementaciones para mejorar la predictibilidad temporal del kernel de Linux. Todas estas soluciones tienen en común la necesidad de modificar o adaptar parte del código del kernel. Esto provoca que sea muy complejo mantener las soluciones actualizadas y disponibles para las últimas versiones del sistema operativo.

Una de las posibles configuraciones software que se pueden plantear a la hora de usar un sistema multinúcleo es el uso en exclusiva de un núcleo por parte de un proceso. Con ello se pretenden evitar las interrupciones en la ejecución provocadas por otros procesos que puedan competir por el tiempo de procesamiento del núcleo.

El presente trabajo tiene como objetivo evaluar alternativas que permitan mejorar la predictibilidad temporal de los sistemas basados en Linux sin necesidad de realizar cambios en el kernel. De este modo se consigue una solución portable y aplicable a sistemas de sobremesa actuales. Para ello se sacará partido de las arquitecturas multinúcleo que son tan comunes hoy en día.

Como ejemplo en el campo de las comunicaciones, la especialización y paralelización de los sistemas de enrutamiento ha proporcionado un incremento muy apreciable en términos de rendimiento y/o latencia [7]. Esta aproximación también se ha usado con éxito en la implementación de controladores OpenFlow [8].

En este trabajo, se procederá estudiar e implementar distintos mecanismos de aislamiento de los núcleos. La intención es ejecutar en exclusiva programas con restricciones temporales en un sistema de propósito general basado en el kernel de Linux.

1.5 Organización de la memoria

La memoria está organizada en los siguientes capítulos, excluyendo el actual:

- El capítulo 2.1 describe los aspectos técnicos y los mecanismos proporcionados por los sistemas Linux de propósito general para el aislamiento de núcleos.
- El capítulo 3 describe las pruebas realizadas para comprobar el alcance de los mecanismos presentados en el capítulo 2.1. Estas pruebas consisten en la medida de diferentes aspectos temporales mediante el uso de benchmarks de `stress` y bajo diferentes configuraciones. También se describe un programa ejecutado en un OBC tipo Raspberry y que a través de los pines de E/S genera una señal cuadrada cuya estabilidad en frecuencia se medirá con diferentes configuraciones.
- El capítulo 4 describe los resultados obtenidos en las pruebas comentadas previamente.
- Finalmente el capítulo 5 contiene las conclusiones y los trabajos futuros.

Además de los capítulos descritos la memoria incluye un anexo con el código fuente de las pruebas realizadas.

Capítulo 2

Estudio teórico

Si no puedes darme poesía, ¿no puedes al menos darme ciencia poética?

Ada Lovelace

2.1 Introducción

Los sistemas operativos basados en el kernel de Linux, a pesar de tener un claro propósito de uso general, han ido evolucionando a lo largo del tiempo en términos de características y comportamiento temporal, de tal modo que han ampliado su capacidad para ser utilizados en entornos más específicos.

El presente capítulo tiene como objetivo estudiar los mecanismos que permitan la predictibilidad temporal de los sistemas basados en Linux sin necesidad de realizar cambios profundos. De este modo se consigue una solución portable y aplicable a cualquier sistema Linux. Para poder alcanzar dicha solución se estudiarán las herramientas que permiten configurar las arquitecturas multinúcleo, tan comunes hoy en día. Básicamente las soluciones que se estudian pretenden el aislamiento de los núcleos del procesador en los que ejecutar las aplicaciones de tiempo real libres o con restricciones de tiempo de las interferencias de las demás aplicaciones y de la mayor parte de las actividades del sistema operativo.

2.2 Marco tecnológico

Como se ha comentado con anterioridad, el objetivo es aislar un núcleo de un ordenador para que se dedique a realizar una tarea en específico. Los aspectos más importantes a tener en cuenta se detallan en las siguientes figuras [9].

De forma general las tres fuentes de perturbación en la ejecución de un programa son:

1. Competición por el tiempo de procesamiento de la CPU con otros procesos de usuario.
2. Competición por el tiempo de procesamiento de la CPU con hilos de ejecución del kernel (Kernel Threads o Kernel Workers (kworker)). Estos son hilos de ejecución lanzados por el kernel para dar respuesta a las llamadas al sistema desde la parte de usuario.
3. Interrupciones.

La figura 2.1 muestra en diferentes colores, azul (núcleo 0), verde (núcleo 1), rosa (núcleo 2) y amarillo (núcleo 3), los componentes software ejecutados por cada núcleo. Como se observa, el núcleo número 1, está ejecutando tres programas de forma simultánea, daemon, RT Application y GP (General Purpose) Application, con lo que la aplicación en tiempo real (RT Application) que se está ejecutando, seguramente se vea interrumpida por el resto de las aplicaciones que se están ejecutando en el mismo núcleo.

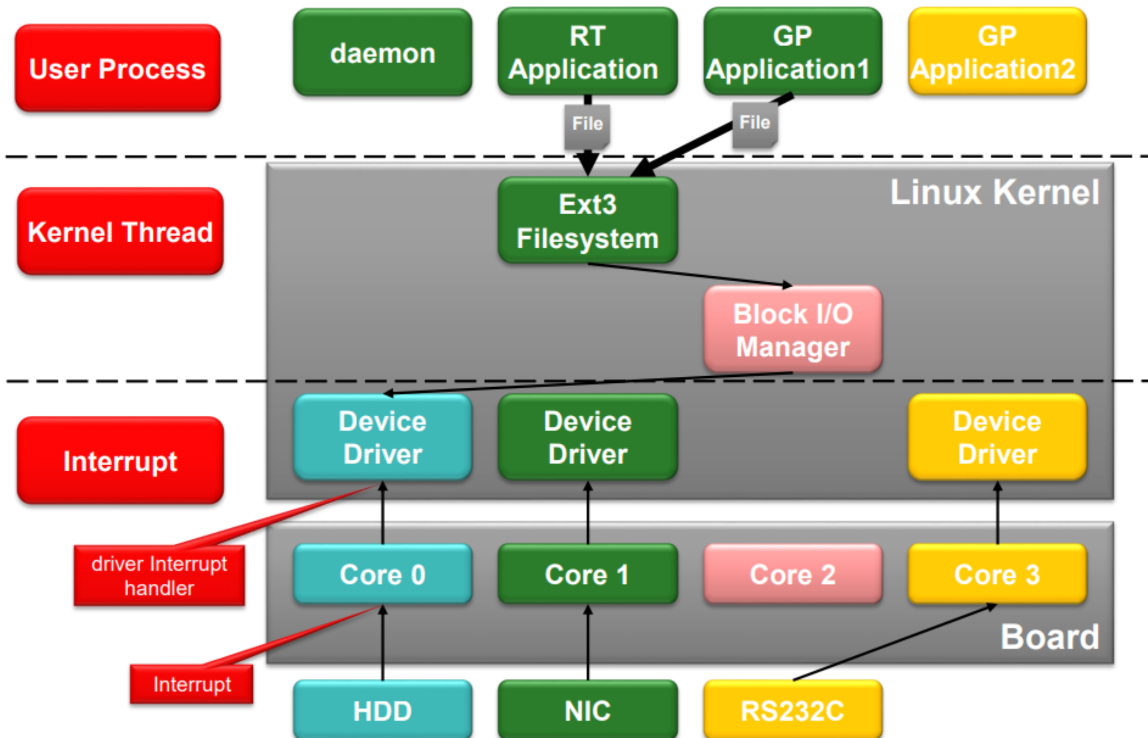

Figura 2.1: Ejecución de un programa en Linux con cuatro núcleos.

Como se muestra en la Figura 2.2, el núcleo 1 está ejecutando varios procesos a la vez. Además, se observa que también depende del núcleo 2 dado que necesita acceder al gestor de E/S en bloque y este a su vez llama al núcleo 0 puesto que este se encarga del acceso al HDD y sus interrupciones.

Con todo esto podemos apreciar que además de estar ejecutando varios procesos a la vez el núcleo 1, depende de otros núcleos que puede que estén ocupados o no, o también que dejen el trabajo que le ha demandado el núcleo 1 para realizar un trabajo con mayor prioridad. A parte, hay que tener en cuenta que el núcleo 1 tendrá que estar pendiente de las interrupciones del NIC (Network Interface Card).

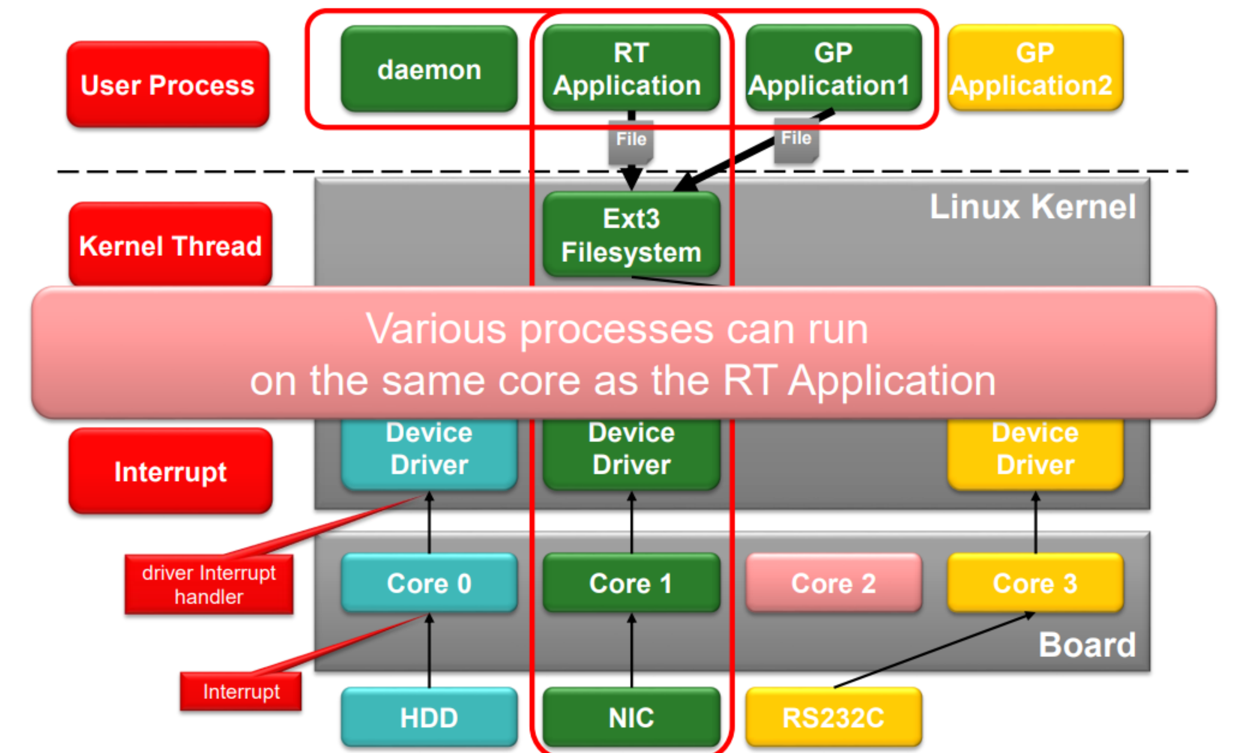

Figura 2.2: Análisis de la ejecución de un programa en Linux con cuatro núcleos.

El objetivo sería realizar un balance de los trabajos asignados a cada núcleo de forma que la tarea con los requisitos temporales más estrictos estuviera sola en un uno de los núcleos. Se puede observar en la Figura 2.3 y 2.4, como se vería de una forma esquematizada el aislamiento de esta aplicación (en este caso de tiempo real) en un núcleo.

En los ejemplos de las figuras, lo que se pretende es que uno de los núcleos (el núcleo 0) tenga toda la carga de trabajo excepto la ejecución de las aplicaciones, las cuales serán ejecutadas por los núcleos que están libres. De esta forma, lo se conseguirá con esto es que si hay interrupciones las tenga solo en el mismo núcleo y esto no genere tanta latencia en la ejecución de la aplicación de tiempo real. Esto es debido a que como un solo núcleo tiene todo el control sobre las interrupciones, será el encargado de gestionar las prioridades de cada aplicación.

En este caso, lo que se realizaría es un orden de prioridades donde por encima de todo se intentaría ejecutar siempre la aplicación en tiempo real (RT Application), para obtener lo anteriormente dicho, una latencia muy baja y así pueda cumplir el objetivo de la aplicación de tiempo real dentro de los rangos de tiempo requeridos y no sufra ninguna interrupción.

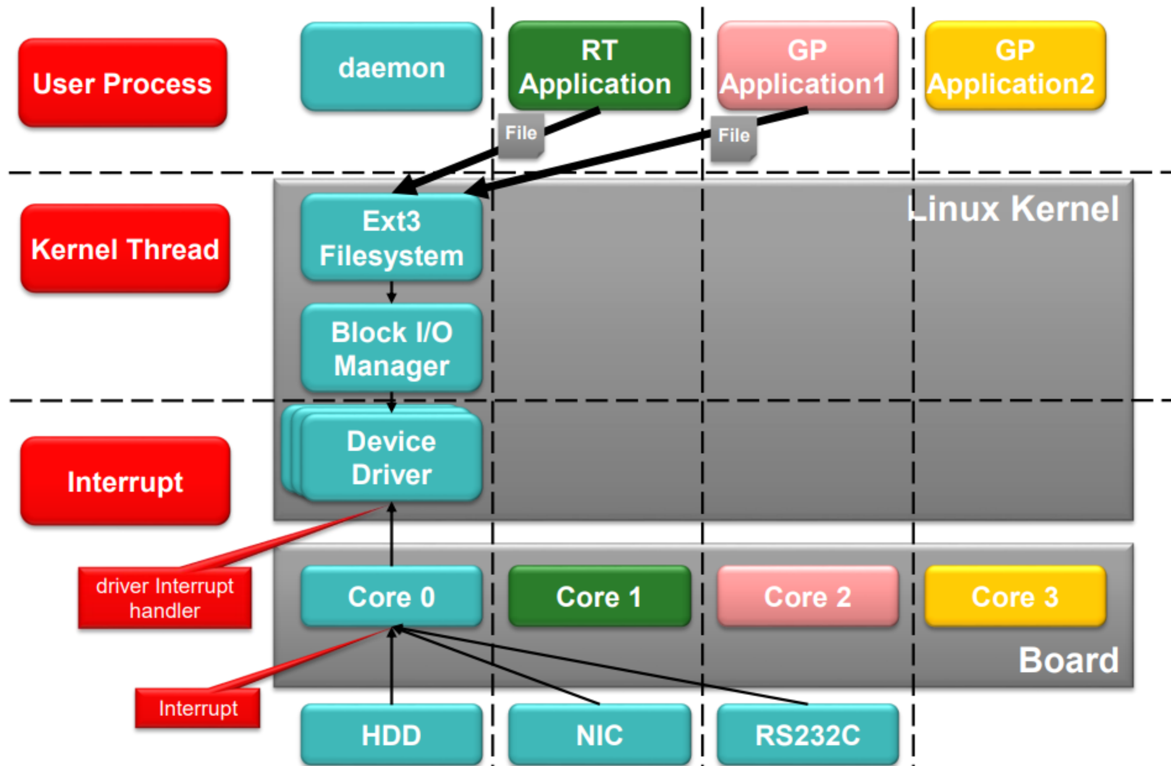

Figura 2.3: Objetivo de ejecución de una aplicación en un núcleo aislado.

Lo que se consigue finalmente, como se ve remarcado en el recuadro rojo, es que la aplicación en tiempo real esté corriendo en un solo núcleo (núcleo 1), el cual no se le envía ningún otro trabajo para no interferir en su ejecución.

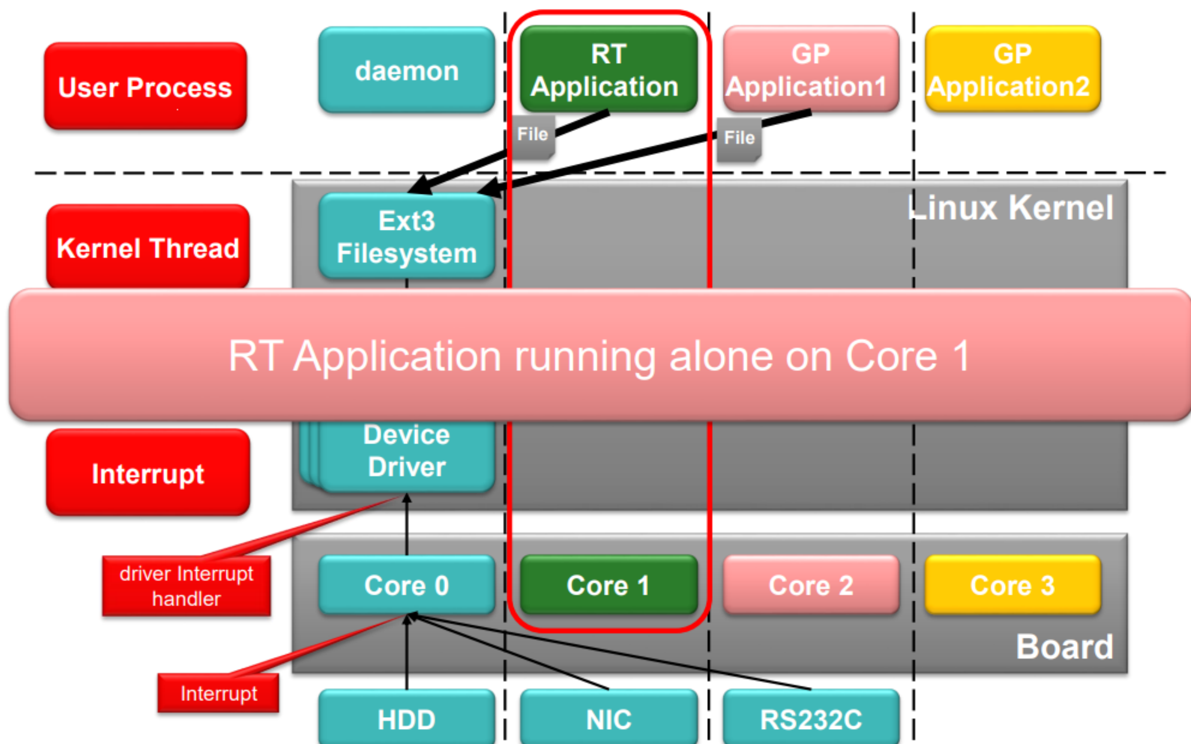

Figura 2.4: Análisis del objetivo del aislamiento de un núcleo.

2.3 Aislamiento de la CPU

En la actualidad, se pueden encontrar dos mecanismos principales de aislamiento de la CPU en Linux. Estos son *isolcpus* y *cpuset*.

2.3.1 Isolcpus

Este mecanismo consiste en la configuración de un parámetro de arranque denominado *isolcpus* que permite establecer una lista de núcleos (o procesadores) para su aislamiento. Una vez realizada la configuración estos no serán tenidos en cuenta por el planificador de Linux ni por el algoritmo de balanceo de carga. Hay que tener en cuenta que los hilos del kernel, los manejadores de interrupciones o procesos/hijos asignados explícitamente mediante su afinidad a este núcleo o procesador aislado, se seguirán ejecutando.

Este tipo de mecanismo tiene la desventaja de que como se ha mencionado anteriormente, es un parámetro de arranque, con lo que se tendrá que realizar su modificación desde el gestor de arranque.

Por otro lado, la propia documentación oficial relativa a los parámetros del kernel desaconseja su uso, indicando que se tendrá que usar el mecanismo de *cpusets*.

2.3.2 CPUSET

Mediante este mecanismo, lo que se puede realizar es la asignación de un conjunto de CPUs y de nodos de memoria a un conjunto de tareas. El acceso no uniforme a memoria (NUMA) es una arquitectura usada en sistemas multiprocesador, en el que el tiempo de acceso a la memoria depende de la ubicación de la memoria en relación con el procesador. En NUMA, un procesador puede acceder a su propia memoria local más rápidamente que a la memoria no local (memoria local de otro procesador o memoria compartida entre procesadores).

Con los *cpuset* se restringe la ubicación de la CPU y la memoria de las tareas a solo los recursos dentro del cpuset actual de una tarea. En los cpuset se pueden construir jerarquías anidadas en las que los recursos son heredados.

A nivel de usuario, se pueden crear y destruir cpusets mediante nombres en el sistema de archivos virtuales de cgroup, gestionar los atributos y permisos de estos cpusets así como qué CPUs y nodos de memoria están asignados a cada cpuset, especificar y consultar a qué cpuset está asignada una tarea, y listar los pids de tareas asignados a un cpuset.

En este estudio, se ha seleccionado cpuset para el aislamiento de la CPU, primero porque como se ha comentado anteriormente, isolcpus se encuentra obsoleto, y segundo porque cpuset permite crear un aislamiento para la CPU estando arrancado el equipo, sin necesidad de reiniciarlo para hacer uso de su funcionamiento.

2.4 Configuración y verificación de CPUSET

Durante este apartado se detalla la configuración que se ha seleccionado para ejecutar más adelante el programa de benchmark. Así, como la verificación del correcto funcionamiento de cpuset.

Para ello, lo primero que hay que llevar a cabo es la creación y montaje del cpuset.

```
root@qui-gon:/home/rasalghul# mkdir /dev/cpuset
root@qui-gon:/home/rasalghul# mount -t cgroup -o cpuset cpuset /dev/cpuset
```

Figura 2.5: Creación y montaje de cpuset.

Como se observa en las líneas anteriores, lo primero que se hace es crear una carpeta, denominada en este caso como `cpuset`, y se ubica en la ruta `/dev`. En segundo, se monta la carpeta, eligiendo el tipo cgroup, con opción de cpuset y anexando el directorio al sistema de ficheros que se encuentra en cpuset.

Ahora, el siguiente paso es ir a la carpeta que se acaba de crear y se crea una nueva. Como se puede observar, esta vez se le ha llamado a la carpeta como `mycpuset`.

```
root@qui-gon:/home/rasalghul# cd /dev/cpuset
root@qui-gon:/dev/cpuset# mkdir mycpuset
```

Figura 2.6: Creación de carpeta nueva dentro de cpuset.

Estando en este punto, se ha de ir a esta nueva carpeta y se establecerá la siguiente configuración:

```
root@qui-gon:/dev/cpuset/mycpuset# echo 0 > cpuset.mems
root@qui-gon:/dev/cpuset/mycpuset# echo 2 > cpuset.cpus
root@qui-gon:/dev/cpuset/mycpuset# echo 0 > cpuset.mems
root@qui-gon:/dev/cpuset/mycpuset# echo 1 > cpuset.cpu_exclusive
root@qui-gon:/dev/cpuset/mycpuset# echo 1 > cpuset.mem_exclusive
root@qui-gon:/dev/cpuset/mycpuset# echo 1 > cpuset.sched_relax_domain_level
```

Figura 2.7: Configuración de los parámetros de cpuset.

Se detallan a continuación la configuración de los parámetros que se han considerada más significativos, una descripción completa se puede encontrar en [10]:

- `cpuset.cpus`. Con este parámetro se le indica al cpuset cuántos núcleos y cuáles van a estar incluidos en este cpuset. En este caso, se ha elegido que se ejecute en el núcleo número 4. Si se tuviesen que elegir varios núcleos, se podrían los números de los núcleos precedidos por comas (0,2,3), o bien, si son núcleos contiguos, de pondría mediante un guión (0-2).

- `cpuset.mems`. Este parámetro sirve para asignar los nodos de memoria que va a utilizar el cpuset que se crea.

- `cpuset.cpu_exclusive`. Se establece el uso de la CPU o CPUs del cpuset como exclusiva o exclusivas.

- `cpuset.mem_exclusive`. Mediante este parámetro se consigue que la cpuset tenga uso exclusivo de sus nodos de memoria. Así mismo, al activar este parámetro se activa de forma implícita, el parámetro `mem_hardwall`, que indica que este cpuset restringe las asignaciones del kernel para páginas, buffers y otros datos comúnmente compartidos por el kernel entre múltiples usuarios.

- `cpuset.sched_relax_domain_level`. En el dominio sched, el planificador migra las tareas de dos formas distintas: balance de carga periódica en tick de reloj, y en el momento de algunos eventos de programación. Como se quiere la menor interrupción posible, así como la menor latencia posible, se ha de poner este valor a *0*, para que utilice ninguna CPU durante la ejecución de la tarea a realizar.

A continuación, se puede observar en la Figura 2.8 con el comando `cset set -l`, que se ha creado el cpuset con las opciones que se han establecido previamente, además de ver si hay más cpusets [11].

```
root@qui-gon:/dev/cpuset/mycpuset# cset set -l
cset:
         Name       CPUs-X    MEMs-X Tasks Subs Path
 ------------ ---------- - ------- - ----- ---- ----------
         root        0-3 y       0 y   814    1 /
     mycpuset          2 y       0 y     0    0 /mycpuset
```

Figura 2.8: Verificación de la creación del cpuset llamado *mycpuset*

Visto que se ha configurado correctamente el cpuset, lo que se tendrá que hacer a continuación es lanzar el siguiente comando "`echo $$ > tasks`" para que el terminal en el que lo hemos lanzado coja la configuración del cpuset y trabaje sobre ella. A partir de aquí, se podrán lanzar las pruebas que se quieran en el cpuset que se ha configurado.

2.5 Afinidad de las interrupciones

Una solicitud de interrupción (IRQ) es una solicitud de servicio solicitado desde el hardware. Cuando se produce una interrupción, se produce un cambio de contexto y el código del kernel recupera el número de IRQ y su lista asociada de Rutinas de servicio de interrupciones registradas (ISR), y a su vez, llama a cada ISR. La ISR realiza el reconocimiento de la interrupción y sitúa un indicador de procesamiento diferido para terminar de procesar la interrupción fuera del contexto de la ISR.

Para balancear la carga que generan las interrupciones, los sistemas operativos basados en Linux poseen un demonio llamado *irqbalance* que ayuda a balancear dicha carga entre los distintos procesadores o núcleos y así aumentar el rendimiento. Aun así, también se encuentra la posibilidad de asignar ciertas interrupciones a procesadores o núcleos específicos. Esta posibilidad es denominada *SMP IRQ affinity* y permite controlar qué núcleos o procesadores ejecutarán las distintas ISR de dicha IRQ producidas en el sistema.

En el directorio */proc/irq*, se encuentra un directorio por cada interrupción, que a su vez contiene un fichero llamado *smp_affinity* [12], en el cuál cabe la posibilidad de cambiar la afinidad de la interrupción modificando una mácara de bits, donde cada uno de ellos corresponde con un núcleo del procesador. Mediante esta máscara se representa qué núcleo o procesador atenderá la interrupción. Si por ejemplo se tiene un procesador con cuatro núcleos y el objetivo es que una interrupción sea atendida solamente por el primer núcleo se escribirá en el fichero el valor hexadecimal 1. En el caso de querer que la interrupción sea atendida por los dos primeros núcleos y por el último de ellos, se ha de escribir el valor hexadecimal 11. Para clarificar estos dos ejemplos, se ilustra en la Figura 2.9 qué valores binarios y hexadecimales debe tener la máscara en función de los núcleos que queramos utilizar.

	Binario	Hexadecimal
CPU 0	0001	1
Máscara	0001	1

	Binario	Hexadecimal
CPU 0	0001	1
CPU 1	0010	2
CPU 3	1000	8
Máscara	1011	11

Figura 2.9: Ejemplo de utilización y configuración de la máscara de bits para determinar qué núcleos del procesador atenderán una interrupción.

2.6 Afinidad de los procesos

La afinidad de procesos (también conocida como «CPU pinning») consiste asignar programas a un núcleo, en lugar de permitir al planificador que decida donde se ejecuta.

En este sentido, el comando `taskset` es de gran utilidad, ya que permite establecer la afinidad de la CPU para a la hora de lanzar un proceso, o bien, cuando este ya se está ejecutando, indicando su PID.

Las principales opciones son:

- -p, --pid Opera en un PID existente y no inicia una nueva tarea
- -c, --cpu-list Especifica una lista numérica de procesadores en lugar de una máscara de bits. La lista puede contener varios elementos, separados por comas y rangos. Por ejemplo, 0,5,7,9-11.

Por ejemplo, el comando `taskset -cp 0,1 9726` muestra y cambia la afinidad del proceso cuyo PID es 9726 a los núcleos 0 y 1.

El resultado sería:

```
pid 9726's current affinity list: 0-3
pid 9726's new affinity list: 0,1
```

Suponiendo un sistema con 4 núcleos, inicialmente el proceso podía ser migrado a cualquiera de ellos (0-3). Después del comando el proceso solo se ejecutará en los núcleos 0 y 1.

Además del comando `taskset`, los procesos pueden ser asignados por programa utilizando llamadas al sistema del tipo `sched_setaffinity`. En este trabajo no se utilizan.

2.7 Aislamiento de núcleos en Windows

Los sistemas Windows ofrecen herramientas semejantes a los descritos para Linux.

- **Primer método**. Se puede configurar en el arranque del sistema (boot.ini) cuántos núcleos se quieren utilizar. Es decir, mediante el parámetro "/numproc" se definen los núcleos que va a utilizar el sistema quedando el resto de los núcleos exentos, y que se puedan usar para aplicaciones concretas. Poniendo un ejemplo, podemos definir /numproc=3, donde el número de procesadores totales es 4, con lo que se quedaría un núcleo el cuál se podría utilizar para aplicaciones en tiempo real.
- **Segundo método**. Se podrá configurar mediante el comando "START" [13]. Con este comando lo que se consigue es lanzar una aplicación en un núcleo o varios específicos. Para ello lo que se tendrá que hacer será pasarle el número en hexadecimal de los núcleos en los que se quiere que se ejecute la aplicación, lo podremos ver en la siguiente Tabla 2.1, que hace referencia a esta documentación de Windows [14].

Tabla 2.1: Tabla de referencia para escoger las CPUs.

CPU ID	**Valor asociado (n)**	**Formula** $(2^n - 1)$	**Afinidad en Hex (h)**
CPU0	1	1	1
CPU1	2	3	3
CPU2	4	7	7
CPU3	8	15	F
CPU4	16	31	1F
CPU5	32	63	3F
CPU6	64	127	7F
CPU7	128	255	FF

Con lo que, si se quisiera utilizar la CPU1 y la CPU4, se tendrían que sumar los valores asociados y pasarlos a hexadecimal. En este ejemplo el comando quedaría de la siguiente forma: start /affinity 12 application.exe

- **Tercer método**. Se puede cambiar la afinidad de una aplicación mediante el uso de "process.ProcessorAffinity" en C# [15]. Esto se puede ejecutar en un script para Windows o también mediante la consola de PowerShell de Windows.

Estos mecanismos bajo Windows no son evaluados en este trabajo y se propondrán como trabajos futuros.

Capítulo 3

Desarrollo

La libertad es poder decir que dos y dos son cuatro. Si se concede esto, todo lo demás vendrá por sus pasos contados.

George Orwell, 1984.

3.1 Introducción

En el capítulo anterior se han descrito los diferentes métodos de aislamiento de núcleos, así como las distintas configuraciones que habría que realizar para implementar estos mecanismos. En este capítulo se va a detallar el desarrollo realizado.

Se mostrarán las distintas configuraciones que han hecho falta para cada uno de nuestros escenarios de prueba, así como desarrollos de scripts para poder llevar a cabo las distintas pruebas.

Primero se empezará describiendo los distintos benchmarks que se han llevado a cabo para observar si el aislamiento del núcleo es lo suficientemente funcional. Una vez descritos estos benchmarks, se hablará del script que se ha desarrollado para la ejecución simultanea de la generación de una gráfica para observar la latencia del benchmark que se va a ejecutar.

También se van a describir el código que se han desarrollado para la Raspberry Pi para la generación de una señal cuadrada a través de los pines de E/S.

3.2 Desarrollo del sistema de experimentación

Para la realización de las pruebas de estrés, se han utilizado los siguientes benchmarks:

- Hackbench. Su función principal es crear un número determinado de pares de entidades programables (ya sean hilos o procesos tradicionales) que se comunican a través de sockets o tuberías y cronometrar el tiempo que tarda cada par en enviar datos de ida y vuelta.
- Stress. Es una herramienta, utilizada para imponer ciertos tipos de estrés computacional al sistema y realizar pruebas de estrés.

- Sysbench. Herramienta de benchmark multihilo basada en LuaJIT[1]. Se utiliza con mayor frecuencia para los puntos de referencia de la base de datos, pero también se puede utilizar para crear cargas de trabajo arbitrariamente complejas que no implican un servidor de base de datos.

Para visualizar la latencia que sufre un sistema durante las pruebas, se ha de utilizar `Cyclictest`, que mide con precisión y repetidamente la diferencia entre el tiempo de activación previsto de un hilo y el momento en que realmente se despierta para proporcionar estadísticas sobre las latencias del sistema.

Con todo lo anterior, se ha creado un shell script, el cual ejecuta de forma simultánea `Cyclictest` y uno de los benchmarks comentados anteriormente generando un gráfico indicando las latencias de los distintos núcleos de la CPU, así como la latencia máxima.

3.3 Latencygraph.sh

3.3.1 Funcionamiento del script

Para la elección del benchmark a ejecutar, se le tiene que pasar como parámetro, teniendo las siguientes opciones:

- **-h**. Ejecutará el comando "hackbench -s 512 -l 1024", el cual enviará 512 bytes por cada mensaje, enviando 1024 mensajes cada par emisor/receptor.
- **-t**. Ejecutará el comando "stress –cpu 8 –io 4 –vm 2 –vm-bytes 128M –timeout 10s", creando 8 procesos intensivos en cada CPU, 4 procesos de entrada y salida, 2 procesos directos a memoria RAM, cada uno de 128 Mb, durante 10 segundos.
- **-h**. Ejecutará el comando "sysbench cpu –cpu-max-prime=50000 run", utilizando el conjunto de test de cpu, donde le indicamos que genere un máximo de 50000 números primos.

Una vez se le ha indicado el benchmark a ejecutar, se selecciona si se van a ejecutar estos benchmarks mediante el uso de taskset o no, e incluso si se quiere que el propio `Cyclictest` se ejecute en el mismo núcleo o en un núcleo distinto al que se está realizando el benchmark. Para elegir esta opción, se ha de introducir el parámetro '-c'. De este modo, el benchmark seleccionado se ejecutará en un núcleo que se ha definido con anterioridad.

Si a continuación se le pasa como parámetro un '1' o un '2', lo que se está seleccionando es, si `Cyclictest` se va a ejecutar en el mismo núcleo que el benchmark (parámetro '1') o si se quiere que se ejecute en un núcleo independiente (parámetro '2'). Se puede observar ver en la Figura 3.1, un diagrama de flujo donde se puede analizar de forma más simplificada el funcionamiento del script.

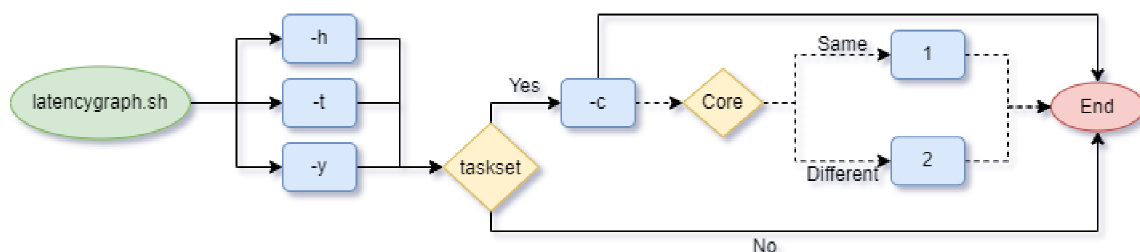

Figura 3.1: Diagrama de flujo del script.

[1]LuaJIT es un compilador justo en tiempo (JIT), compila el código durante la ejecución del programa, para el lenguaje de programación Lua. Lua es un lenguaje de programación potente, dinámico y ligero. Puede integrarse o utilizarse como lenguaje independiente de propósito general [16].

3.3.2 Código del script

En este apartado se procede a detallar el script que se encuentra en el Apéndice A para una mejor compresión.

3.3.2.1 Variables

En la primera parte del script (Figura 3.2) se puede observar la definición de las variables que utilizará este script.

```
######################## VARIABLES ########################

RUTA="/home/rasalghul/TFG/graphic"
CYCL="cyclictest -l100000 -m -Sp90 -i200 -h400 -q"
HACK="hackbench -s 512 -l 1024 -P"
STRESS="stress --cpu 8 --io 4 --vm 2 --vm-bytes 128M --timeout 10s"
SYSB="sysbench cpu --cpu-max-prime=50000 run"
NAME=""
flag=0

###########################################################
```

Figura 3.2: Variables del script.

La primera indica dónde se almacenarán los gráficos generados por el script. Las siguientes variables hasta `SYSB`, serán utilizadas para ejecutar los benchmarks explicados con anterioridad. La variable `NAME`, es utilizada para la generación de los distintos nombres al utilizar distintos benchmarks, para así no confundir las gráficas que vaya generando, y poder realizar varias pruebas con distintos benchmarks y distintas opciones sin necesidad de tener que cambiarle el nombre de forma manual (quedará más claro durante las siguientes explicaciones del código). La última variable, llamada `flag`, e inicializada a 0, está indicando el modo en el que se va a ejecutar el comando `Cyclictest`.

3.3.2.2 Funciones

Una vez explicadas las variables que se han creado, se procede a detallar la función que se ha creado. Esta, está relacionada con el comando `Cyclictest`. Como se observa en la Figura 3.3, se contemplan tres opciones. La primera de ellas es si el parámetro que se le pasa es un 1, en dicho caso, ejecutará el comando `Cyclictest` con taskset, en el segundo núcleo del procesador. También se puede ver que ya se hace referencia a la variable `NAME`, a la cual se le añade `-tg`, indicando que se ha ejecutado `Cyclictest` en el mismo núcleo que el benchmark (together).

La segunda opción que se encuentra es si cuando se le pasa por parámetro el número 2. En este caso, lo que realiza es una ejecución del comando `Cyclictest` en el tercer núcleo del procesador, usando taskset. Ahora, en vez de añadir la etiqueta `-tg`, añade la etiqueta `-sp`, dado que se ha ejecutado `Cyclictest` en un núcleo distinto al núcleo donde se está ejecutando el benchmark (separate).

En la última opción, lo que se encuentra es una opción por defecto. En el caso, en el que no se le pase ningún parámetro o un número distinto al 1 o al 2, lo que hará será ejecutar el comando `Cyclictest` sin indicarle ningún núcleo en concreto donde ejecutarse, con lo que se repartirán la tarea entre los distintos núcleos. Por esta razón, la etiqueta que se añade al nombre es `-s` (simple).

```
######################## FUNCTIONS ########################

cyclic_opt(){
    if [ "$1" == 1 ]
    then
        taskset -c 2 $CYCL >$RUTA/output
        NAME="${NAME}-tg"
    elif [ "$1" == 2 ]
    then
        taskset -c 3 $CYCL >$RUTA/output
        NAME="${NAME}-sp"
    else
        $CYCL >$RUTA/output
        NAME="${NAME}-s"
    fi
}

###########################################################
```

Figura 3.3: Funciones del script.

3.3.2.3 Main

En este subapartado, se pasará a detallar el cuerpo del script. Para empezar, se ejecuta un switch case para seleccionar el benchmark y las opciones que se han comentado con anterioridad, sin que muestre por pantalla la salida de cada de uno de estos.

- Primer case. Se comprueba si se ha seleccionado algún benchmark, en el caso de que no haya seleccionado ninguno, saldrá y ejecutará solamente el `Cyclictest`. Si ha seleccionado alguno, pasará al siguiente case.

- Segundo case. En este caso, lo que hará será comprobar si ha seleccionado que se ejecute con tasktset o no. Por defecto no utilizará taskset.

```
# 0. Choice the test would be execute.

case "$1" in
    -h) case "$2" in
            -c) taskset -c 2 $HACK >> /dev/null 2>&1 &
                NAME="T-hack"
                ;;

            *) $HACK >> /dev/null 2>&1 &
               NAME="Hack"
               ;;
        esac;;
    -t) case "$2" in
            -c) taskset -c 2 $STRESS >> /dev/null 2>&1 &
                NAME="T-str"
                ;;
            *) $STRESS >> /dev/null 2>&1 &
                NAME="Str"
                ;;
        esac;;
    -y) case "$2" in
            -c) taskset -c 2 $SYSB >> /dev/null 2>&1 &
                NAME="T-sys"
                ;;
            *) $SYSB >> /dev/null 2>&1 &
                NAME="Sys"
                ;;
        esac;;
esac
```

Figura 3.4: Switch case para la elección del benchmark a ejecutar.

A continuación, se ejecuta la función de `cyclc`, pasándole por parámetro el último parámetro que el usuario. Seguido a esto, lo que se hace es obtener la máxima latencia que ha tenido la ejecución. Se buscan las líneas de datos, se eliminan las líneas vacías y se crea un separador de campos común. Se obtienen el número de cores que utiliza el ordenador, y se crean dos columnas con la latencia y los valores de frecuencia para cada núcleo, así, después en la gráfica se podrán obtener las estadísticas de cada uno de los núcleos.

```
# 1. Execution of Cyclictest
cyclic_opt $3

# 2. Get maximum latency
max=`grep "Max Latencies" $RUTA/output | tr " " "\n" | sort -n | tail -1 | sed s/^0*//`

# 3. Grep data lines, remove empty lines and create a common field separator
grep -v -e "^#" -e "^$" $RUTA/output | tr " " "\t" >$RUTA/histogram

# 4. Set the number of cores
cores=`nproc`

# 5. Create two-column data sets with latency classes and frequency values for each core
for i in `seq 1 $cores`
do
  column=`expr $i + 1`
  cut -f1,$column $RUTA/histogram >$RUTA/histogram$i
done
```

Figura 3.5: Bloque de comandos para la obtención de los datos de Cyclictest.

Ahora se construye el encabezado, títulos y leyenda de la gráfica tal y como se observan en las figuras 3.6 y 3.7. Cabe destacar, que para la realización de las gráficas se ha tomado como referencia un script ya generado [17].

```
# 6. Create plot command header
echo -n -e "set title \"Latency plot\"\n\
set terminal png\n\
set xlabel \"Latency (us), max $max us\"\n\
set logscale y\n\
set xrange [0:*]\n\
set yrange [0.8:*]\n\
set ylabel \"Number of latency samples\"\n\
set output \"$RUTA/$NAME.png\"\n\
plot " >$RUTA/plotcmd
```

Figura 3.6: Título y nombres de los ejes de la gráfica.

```
# 7. Append plot command data references
for i in `seq 1 $cores`
do
  if test $i != 1
  then
    echo -n ", " >>$RUTA/plotcmd
  fi
  cpuno=`expr $i - 1`
  if test $cpuno -lt 10
  then
    title=" CPU$cpuno"
   else
    title="CPU$cpuno"
  fi
  echo -n "\"$RUTA/histogram$i\" using 1:2 title
        \"$title\" with histeps" >>$RUTA/plotcmd
done
```

Figura 3.7: Anexión de los datos para la leyenda.

Para finalizar la explicación del script, se encuentran las dos últimas líneas que se ven en la Figura 3.8.

```
# 8. Execute plot command
gnuplot -p $RUTA/plotcmd

find $RUTA -type f -not -name '*.png' -delete
```

Figura 3.8: Generación de gráfica y eliminación de archivos temporales.

La primera realiza la función de generar la gráfica con los datos que se han almacenado y la segunda, limpia la ruta que se ha utilizado, dejando únicamente la gráfica con extensión PNG que se ha generado.

3.4 Raspberry

La otra prueba realizada ha consistido en la codificación de un programa ejecutado en OBC tipo Raspberry Pi Model 3. Este programa, a través de los puertos de E/S genera una señal cuadrada de 5 Mhz de frecuencia. La prueba consiste en analizar la estabilidad temporal de esta señal dependiendo del tipo de configuración elegida para su ejecución.

La frecuencia de 5 Mhz se ha seleccionado después de diferentes pruebas ya que es la frecuencia más alta que se ha podido conseguir con una buena relación de aspecto en cuanto a los flancos generados.

A continuación, se describirá los códigos fuente que se encuentran en el Apéndice A.

El primero de ellos se trata del código enable_ccr.c. Este código sirve para cargar un módulo del kernel de Linux para poder utilizar los registros referentes a los contadores de prestaciones presentes en

la Raspberry. De forma general estos contadores no están disponibles para la parte de usuario por lo que deben ser habilitados mediante un módulo kernel.

3.4.1 enable_ccr.c

Antes de empezar, tenemos que tener claro la instrucción MCR utilizada dentro del conjunto de instrucciones de ensamblador que utiliza ARM. Esta es una instrucción genérica de interoperabilidad del coprocesador, pudiendo mover registros desde el ARM a un coprocesador. La sintaxis utilizada para el correcto funcionamiento sería la siguiente [18]:

```
MCR{cond} coproc, #opcode1, Rt, CRn, CRm{, #opcode2}
```

Una vez que ya sabemos la sintaxis y para qué sirve la instrucción MCR, podemos estructurar el código en tres partes:

- Realización de una función para la habilitación de los registros referentes a los contadores (enable_ccr(void *info)).
- Inicio de la carga del módulo de kernel (init_module(void)).
- Finalización del módulo kernel (cleanup_module(void)).

Primero, se va a observar lo que sucede en la primera función llamada enable_ccr(void *info). Dentro de esta función se encuentra en 3 ocasiones el comando asm volatile, como se observa en la Figura 3.9, y se procede a detallar a continuación.

```c
void enable_ccr(void *info) {
  // Set the User Enable register, bit 0
  asm volatile ("mcr p15, 0, %0, c9, c14, 0" :: "r" (1));

  // Enable all counters in the PNMC control-register
  asm volatile ("MCR p15, 0, %0, c9, c12, 0\t\n" :: "r"(1));

  // Enable cycle counter specifically
  asm volatile ("MCR p15, 0, %0, c9, c12, 1\t\n" :: "r"(0x80000000));
}
```

Figura 3.9: Función enable_ccr(void *info).

Como se explicó con anterioridad la sintaxis de la instrucción MCR, se puede entrar en profundidad a explicar las tres ocasiones en las que sale. En las 3 instrucciones se hace referencia a p15, que corresponde al coprocesador 15 (Figura 3.10). Los registros del sistema del CP15 proporcionan información de control y estado para las funciones implementadas en el procesador. Las principales funciones de los registros del sistema CP15 son [1, Capítulo B6.1.71]:

- Control y configuración general del sistema.
- Configuración y gestión de la unidad de gestión de memoria (MMU).
- Configuración y gestión de la caché.

- Virtualización y seguridad.
- Control del rendimiento del sistema.

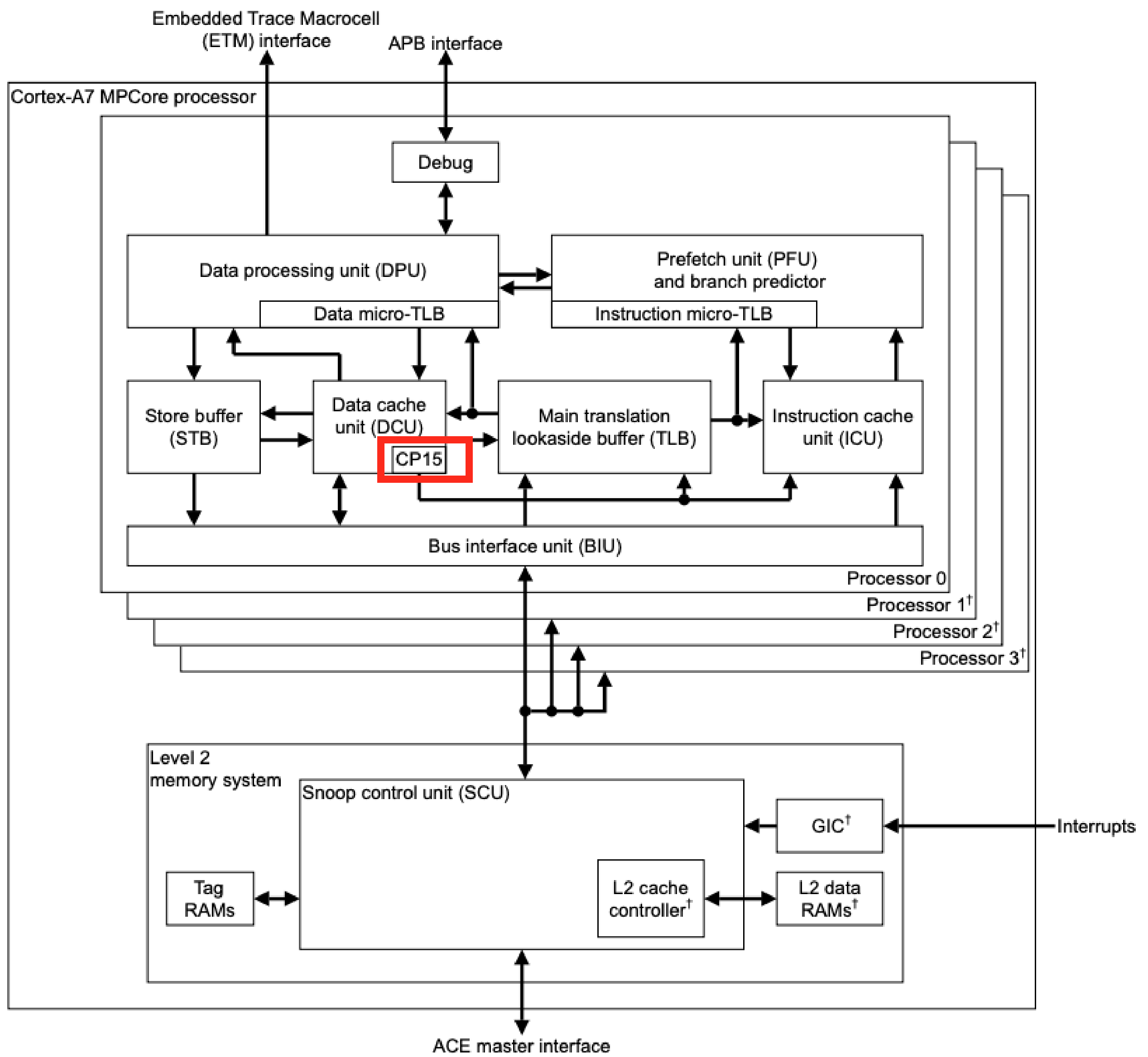

Figura 3.10: Diagrama de nivel superior del procesador Cortex-A7 MPCore.

Continuando con las sentencias de la función, la primera de ellas se interpreta mejor si se lee mediante la siguiente Tabla 3.1:

Tabla 3.1: Habilitar registro de usuario.

CRn	Op1	CRm	Op2	Name	Reset	Description
c9	0	c14	0	PMUSERENR	0x00000000	User Enable Register

Lo que hacemos con las opciones de `%0` y `:: "r"(1)`, es escribir en el primer bit un 1, habilitando el modo usuario como se observa en la Figura 3.11.

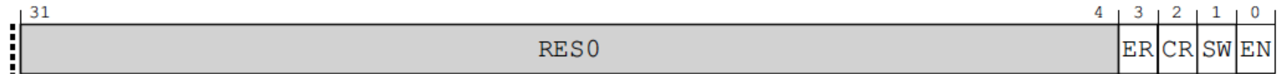

Figura 3.11: Bits de asignación para PMUSERENR [1, Capítulo D13.5.17].

Mediante la segunda instrucción (que se observa con mayor detalle en la Tabla 3.2) lo que se consigue es habilitar todos los contadores del registro de control del monitor de rendimiento (Performance Monitor Control Register, PMCR), el cual controla y configura todos los contadores. Estos registros son de lectura/escritura, tiene dos estados "Seguro y No-Seguro" y son accesibles desde el PL1 o más altos. Pero en esta ocasión, como con anterioridad hemos habilitado el modo usuario, sí que se puede escribir ya en estos registros. Se detalla en la siguiente Figura 3.12 la asignación de los bits del PMCR.

Tabla 3.2: Habilitación del PMCR.

CRn	Op1	CRm	Op2	Name	Reset	Description
c9	0	c12	0	PMCR	0x41072000	PMCR explicado con anterioridad

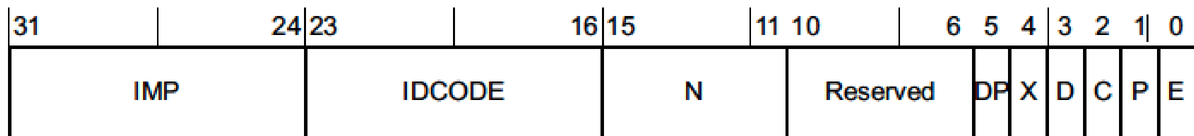

Figura 3.12: Asignación de bits para el PMCR.

Como se observa, lo que se realiza es poner el bit `0` que hace referencia a "E" (Enable bit) a 1, habilitando todos los contadores.

Respecto a la última instrucción (véase en la Tabla 3.3), lo que se realiza es acceder al registro de habilitación del recuento de los monitores de rendimiento (Performance Monitors Count Enable Set register, PMCNTENSET) y habilitarlo, ya que al establecer `0x80000000`, lo que se hace es poner a 1 el bit C, que hace referencia al registro de recuento de ciclos de los monitores de rendimiento (Performance Monitors Cycle Count Register, PMCCNTR).

Tabla 3.3: Habilitar el recuento de los monitores de rendimiento.

CRn	Op1	CRm	Op2	Name	Reset	Description
c9	0	c12	1	PMNCNTENSET	UNK	Count Enable Set Register, explicado con anterioridad

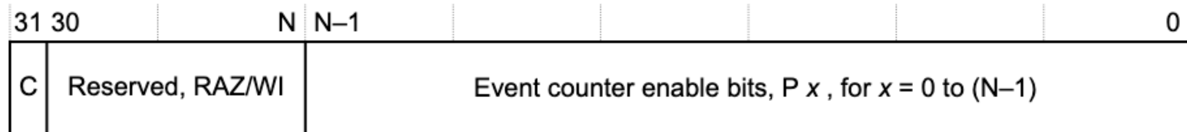

Figura 3.13: Asignación de bits para el PMCNTENSET.

Se continua ahora con la función init_module(void) (Figura 3.14), cuya función es iniciar la carga del módulo, con sus distintas operaciones dentro de la función. Lo primero que se hace es llamar a la función `on_each_cpu()`, la cual, consultado con el kernel actual 5.16.10, se encuentra definida en el archivo "spm.h" [19] y lo que hace es ejecutar una función que se le pasa por parámetro. A esta función se le pasa primero la función que se quiere ejecutar, en este caso la función `enable_ccr`, después el segundo parámetro es un puntero, el cual es nulo para este caso, y a continuación se le indica si se quiere esperar entre la ejecución de la función entre los distintos procesadores, que se define como 0 para qué no esperen, ya que son procesos independientes para esta situación.

```
int init_module(void) {
  // Each cpu has its own set of registers
  on_each_cpu(enable_ccr,NULL,0);
  printk (KERN_INFO "Userspace access to CCR enabled\n");
  return 0;
}
```

Figura 3.14: Función init_module(void).

Las dos últimas líneas de esta función hacen referencia a, la primera imprimir por pantalla que el acceso del espacio de usuario ha sido habilitado para el CCR, y la segunda simplemente indica el retorno de 0, indicando que todo ha salido bien.

Para finalizar, se tiene la función cleanup_module(void). Esta función está vacía ya que es la forma de indicar que ya ha finalizado la carga e implementación del módulo kernel que se ha querido aplicar.

3.4.2 signal.c

Este código genera la señal cuadrada. Para ello activa/desactiva uno de los pines de E/S realizando esperas activas mediante la lectura del contador de prestaciones.

Para empezar, se incluyen las librerías necesarias (Figura 3.15a) y se especifican todas las variables y pines propios de la Raspberry, tal y como se muestra en la Figura 3.15b.

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <stdint.h>
4
5   #include <fcntl.h>
6   #include <sys/mman.h>
7   #include <sys/stat.h>
8   #include <unistd.h>
9
```

(a) Librerías necesarias.

```
10  #ifndef PI_VERSION
11  #  define PI_VERSION 3
12  #endif
13
14  #define BCM2708_PI1_PERI_BASE  0x20000000
15  #define BCM2709_PI2_PERI_BASE  0x3F000000
16  #define BCM2711_PI4_PERI_BASE  0xFE000000
17
18  // --- General, Pi-specific setup.
19  #if PI_VERSION == 1
20  #  define PERI_BASE BCM2708_PI1_PERI_BASE
21  #elif PI_VERSION == 2 || PI_VERSION == 3
22  #  define PERI_BASE BCM2709_PI2_PERI_BASE
23  #else
24  #  define PERI_BASE BCM2711_PI4_PERI_BASE
25  #endif
26
27  // ---- GPIO specific defines
28  #define GPIO_REGISTER_BASE 0x200000
29  #define GPIO_SET_OFFSET 0x1C
30  #define GPIO_CLR_OFFSET 0x28
31  #define PHYSICAL_GPIO_BUS (0x7E000000 + GPIO_REGISTER_BASE)
32
33  #define TIME_LOW  60000
34  #define TIME_HIGH 60000
35
36  //const int GPIO_OUT=22;  // pin 15
37  #define TOGGLE_GPIO 22
38
39  #define PAGE_SIZE 4096
```

(b) Definición de variables y asignación de pines.

Figura 3.15: Preparación del entorno de la Raspberry.

Cabe destacar las constantes GPIO_SET_OFFSET y GPIO_CLR_OFFSET que contienen los desplazamientos de los registros de control que permite poner a '1' (SET) o poner a '0' (CLR) los pines de E/S.

Los valores TIME_LOW y TIME_HIGH definen los `ticks` por defecto que definen el tiempo a nivel alto o a nivel bajo de la señal cuadrada. Estos valores podrán ser modificados por la línea de comandos, como se muestra a continuación 3.16.

```
81  int main(int argc, char **argv)
82  {
83  uint32_t time_low, time_high;
84
85  time_high = TIME_HIGH;
86  time_low  = TIME_LOW;
87
88  if (argc == 3 )
89  {
90    time_low  = atoi( argv[1] );
91    time_high = atoi( argv[2] );
92  }
```

Figura 3.16: Definición y asignación de las variables time_high y time_low.

A continuación, se configura la Entrada/Salida de Propósito General (GPIO, General Purpose Input/Output) de la Raspberry para generar la señal. Para ello, se han usado las siguientes declaraciones.

```
94   // Prepare GPIO
95   volatile uint32_t *gpio_port = mmap_bcm_register(GPIO_REGISTER_BASE);
96   initialize_gpio_for_output(gpio_port, TOGGLE_GPIO);
97   volatile uint32_t *set_reg = gpio_port + (GPIO_SET_OFFSET / sizeof(uint32_t));
98   volatile uint32_t *clr_reg = gpio_port + (GPIO_CLR_OFFSET / sizeof(uint32_t));
```

Figura 3.17: Preparación de GPIO.

La primera función es `mmap_bcm_register()`, Figura 3.18. A esta función se le pasa mediante parámetro el desplazamiento del registro al que desea acceder y devuelve un puntero al registro mapeado en memoria. Para ello se mapea la dirección física del conjunto de registros en el espacio de direcciones del programa mediante la llamada `mmap` y retorna un puntero a ese conjunto de registros. El pseudo archivo `/dev/mem` proporciona un acceso en modo ROOT a la memoria física de la Raspberry.

```
41  // Return a pointer to a periphery subsystem register.
42  static void *mmap_bcm_register(off_t register_offset) {
43    const off_t base = PERI_BASE;
44
45    int mem_fd;
46    if ((mem_fd = open("/dev/mem", O_RDWR|O_SYNC) ) < 0) {
47      perror("can't open /dev/mem: ");
48      fprintf(stderr, "You need to run this as root!\n");
49      return NULL;
50    }
51
52    uint32_t *result =
53      (uint32_t*) mmap(NULL,                  // Any adddress in our space will do
54                       PAGE_SIZE,
55                       PROT_READ|PROT_WRITE,  // Enable r/w on GPIO registers.
56                       MAP_SHARED,
57                       mem_fd,                // File to map
58                       base + register_offset // Offset to bcm register
59                       );
60    close(mem_fd);
61
62    if (result == MAP_FAILED) {
63      fprintf(stderr, "mmap error %p\n", result);
64      return NULL;
65    }
66    return result;
67  }
```

Figura 3.18: Función mmap_bcm_register().

La función `initialize_gpio_for_output()`, se utiliza para configurar un registro de la Raspberry como salida. En este caso, se le pasa por parámetro el puntero que apunta al mapa de registros que se había generado anteriormente, junto con el desplazamiento del GPIO 22. La estructura de este registro se observa en la Figura 3.20, extraída del apartado "5.2. Register View" del manual [20]. Lo que se hace es seleccionar el registro GPFSEL2, que viene dado por el resultado de la expresión `gpio_registerset+(bit/10)`, ya que `gpio_registerset` apunta a los registros (GPFSEL#) junto con la división resultante, cuyo resultado es 2. Una vez se escoge el registro, lo habilitamos como salida poniendo a uno su primer bit, para ello se usa el resto de la expresión, `|= (1<<((bit%10)*3))`, en

la cual se asigna a uno el primer bit del campo 2 del registro GPFSEL2. Cada campo tiene un tamaño de 3 bits.

```
69  void initialize_gpio_for_output(volatile uint32_t *gpio_registerset, int bit) {
70      *(gpio_registerset+(bit/10)) |=  (1<<((bit%10)*3));  // set as output.
71  }
```

Figura 3.19: Función initialize_gpio_for_output().

GPFSEL2 Register

Bits	Name	Description	Type	Reset
31:30	Reserved.	-	-	-
29:27	FSEL29	FSEL29 - Function Select 29 000 = GPIO Pin 29 is an input 001 = GPIO Pin 29 is an output 100 = GPIO Pin 29 takes alternate function 0 101 = GPIO Pin 29 takes alternate function 1 110 = GPIO Pin 29 takes alternate function 2 111 = GPIO Pin 29 takes alternate function 3 011 = GPIO Pin 29 takes alternate function 4 010 = GPIO Pin 29 takes alternate function 5	RW	0x0
26:24	FSEL28	FSEL28 - Function Select 28	RW	0x0
23:21	FSEL27	FSEL27 - Function Select 27	RW	0x0
20:18	FSEL26	FSEL26 - Function Select 26	RW	0x0
17:15	FSEL25	FSEL25 - Function Select 25	RW	0x0
14:12	FSEL24	FSEL24 - Function Select 24	RW	0x0
11:9	FSEL23	FSEL23 - Function Select 23	RW	0x0
8:6	FSEL22	FSEL22 - Function Select 22	RW	0x0
5:3	FSEL21	FSEL21 - Function Select 21	RW	0x0
2:0	FSEL20	FSEL20 - Function Select 20	RW	0x0

Figura 3.20: Detalle del registro de configuración GPSEL2.

Una vez configurado el pin como salida se define un contador `t0` e iniciamos un bucle infinito para la generación de la señal.

```
101    while( 1 )
102    {
103      // Enable counter and reset it
104      __asm__ volatile ("MCR p15, 0, %0, c9, c12, 0\t\n" :: "r"(4+1));
105
106      *clr_reg = (1<<TOGGLE_GPIO);
107
108      do {
109        t0 = ccnt_read();
110      } while(time_low > t0);
111
112      // Enable counter and reset it
113      __asm__ volatile ("MCR p15, 0, %0, c9, c12, 0\t\n" :: "r"(4+1));
114
115      *set_reg = (1<<TOGGLE_GPIO);
116
117      do {
118        t0 = ccnt_read();
119      } while(time_high > t0);
120    }
121    }
```

Figura 3.21: Bucle para la lectura de los valores del contador.

Dentro del bucle, se resetea el contador en cada iteración y se pone a cero el pin de E/S mediante `clr_reg`. Una vez hecho esto, se ejecuta un bucle en el que se esperará hasta que `t0` alcance el tiempo a nivel bajo que se ha definido. Para ello, se actualiza la variable ejecutando la función `ccnt_read()` (Figura 3.22) que lo que hace es leer los monitores de rendimiento del registro de recuento de ciclos (Performance Monitors Cycle Count Register, PMCCNTR). Es decir, devuelve el valor que hay en el PMCCNTR al acceder al registro de control 9 y una vez aquí al 13 como se muestra en la siguiente Tabla 3.4.

Tabla 3.4: Acceso al PMCCNTR.

CRn	Op1	CRm	Op2	Name	Reset	Description
c9	0	c13	0	PMCCNTR	UNK	Cycle Count Register

```
73    static inline uint32_t ccnt_read (void)
74    {
75      uint32_t cc = 0;
76      __asm__ volatile ("mrc p15, 0, %0, c9, c13, 0" : "=r" (cc));
77      return cc;
78    }
```

Figura 3.22: Función para la lectura del PMCCNTR.

Cuando se sobrepasa el tiempo de bajada y se sale del bucle, se resetea el contador de nuevo y se pone a uno el pin de E/S mediante `set_reg` A continuación se ejecuta el bucle de espera mientras el contador no alcance el valor indicado.

Como prueba, este programa se ejecutará en un entorno sin ningún tipo de aislamiento y en un entorno donde se ejecuta en un núcleo en exclusiva, capturando en ambos casos un segmento temporal de la salida mediante un analizador lógico. En el siguiente capítulo se muestran los resultados obtenidos en las pruebas descritas.

Capítulo 4

Resultados

La boca puede mentir, pero la mueca del momento revela la verdad.

Friedrich Nietzsche.

4.1 Introducción

En el capítulo anterior, se ha detallado el desarrollo que se ha llevado a cabo para Ubuntu, como el desarrollo que se ha llevado a cabo en Raspberry Pi.

Ahora se van a analizar los datos que se han obtenido con cada uno de los benchmarks en los distintos entornos. Primero se hablará del entorno experimental de Ubuntu, en el cual se han desarrollado los benchmarks de hackbench, stress y sysbench.

Dentro de cada uno de estos benchmarks, se han realizado pruebas distintas como ejecución del benchmark de forma nominal, aislando el núcleo, separando el generador de la gráfica del propio benchmark,...

Por otro lado, se tienen los resultados de las pruebas realizadas en el entorno de la Raspberry Pi, en el cual las pruebas han consistido en aislar o no uno de los núcleos para analizar la estabilidad de la señal generada a través de los pines de E/S.

4.2 Entorno experimental

4.2.1 Ubuntu

Este estudio se ha realizado utilizando la distribución Ubuntu, versión 21.10, del sistema operativo Linux, cuya versión de kernel es “5.13.0-40-generic”. Este apartado se encuentra divido en distintos grupos de pruebas de estrés (mediante el script “Latencygraph.sh”) los cuales se han realizado bajo distintas condiciones de aislamiento del núcleo a través de los distintos mecanismos que han sido explicados con anterioridad.

Los resultados de las pruebas de estrés realizadas, así como sus análisis, se pueden ver en las siguientes secciones.

4.2.1.1 Hackbench

Empezamos analizando los datos obtenidos al ejecutar la prueba de estrés llamada “Hackbench”.

Lo primero que se hizo fue ejecutar el script de forma nominal, es decir, como un usuario normal, sin tener en cuenta la carga que tenían los núcleos (Figura 4.1). En este caso, se puede apreciar una carga distribuida entre los distintos núcleos del procesador además de que se observa que el tiempo de latencia se distribuye a lo largo del eje X, lo que indica que alarga el tiempo de ejecución dado que existen interrupciones en los distintos núcleos, lo que hace que la tarea finalice más tarde.

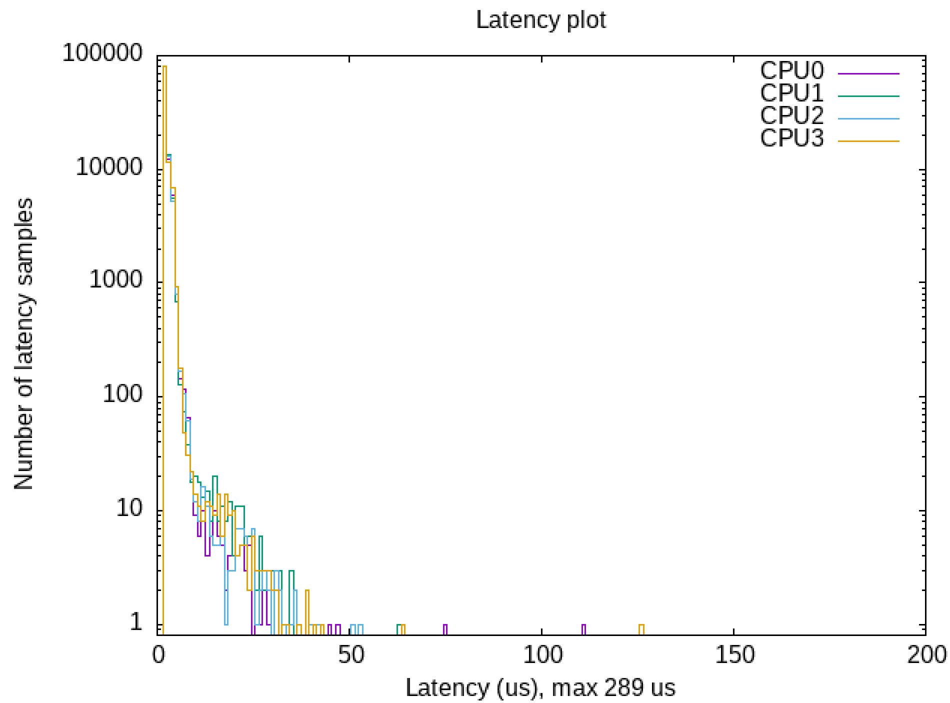

Figura 4.1: Hackbench simple.

4.2.1.2 Pruebas con Taskset

A continuación, se pasa a ejecutar el script, pero con el modo taskset, como se ha visto anteriormente en las opciones del script (Capítulo 3.3.1, Figura 3.1). En la primera figura relacionada con el taskset (Figura 4.2) se observa una pequeña modificación respecto a cuándo se ha ejecutado el script de forma nominal, y es que ahora se puede llegar a apreciar un ligero aumento de la compactación de las tareas.

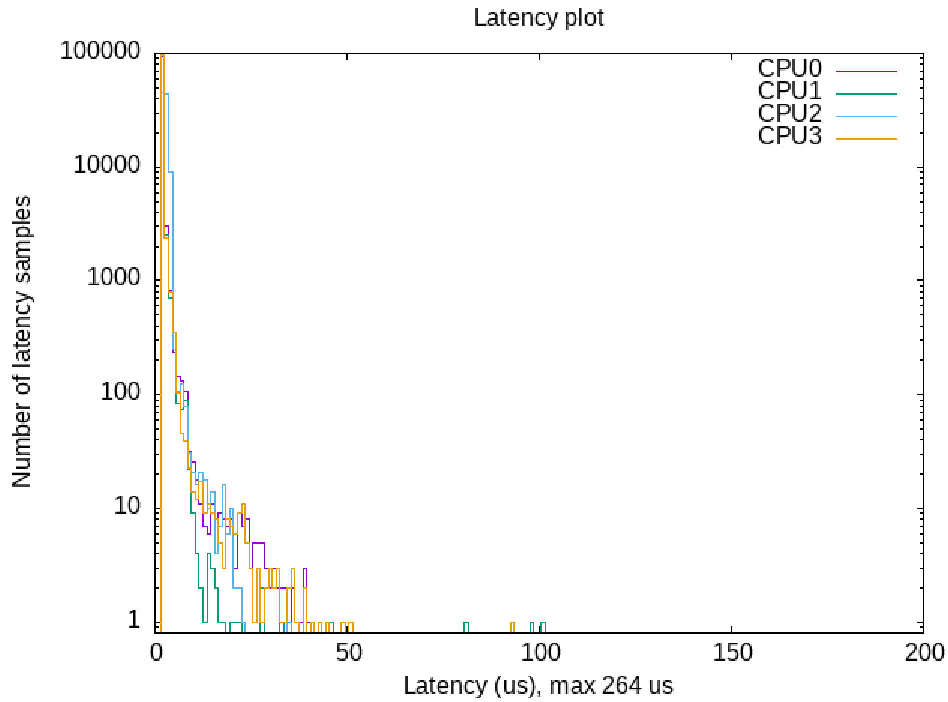

Figura 4.2: Hackbench con T simple.

En la segunda figura (Figura 4.3), al estar ejecutando el cyclictest para la generación de gráficos en un núcleo en específico obtenemos solo la señal del núcleo en el que estamos ejecutando la prueba de estrés. Como se observa, no hay apenas interrupciones de otros núcleos ya que se observa más compactación, además de que la latencia máxima se ha reducido hasta cuatro veces en comparación con la ejecución nominal de la prueba de estrés.

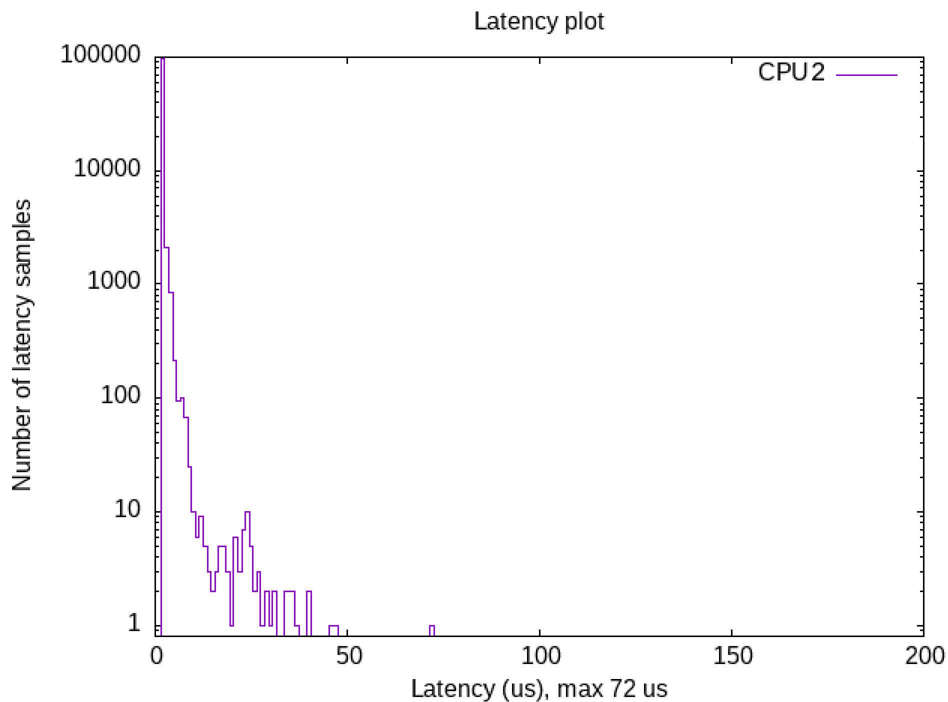

Figura 4.3: Hackbench con T juntos.

Por último, vemos una ejecución compacta, prácticamente sin interrupciones ajenas a la ejecución del script (Figura 4.4). Esto es debido a que la generación de la gráfica usando cyclictest y la prueba de estrés se han ejecutado en núcleos distintos, pero aislados mediante la ejecución del taskset. Esto lo que con lleva es una ejecución lineal de las pruebas y la no interrupción ni bloqueo de recursos entre unas pruebas y otras.

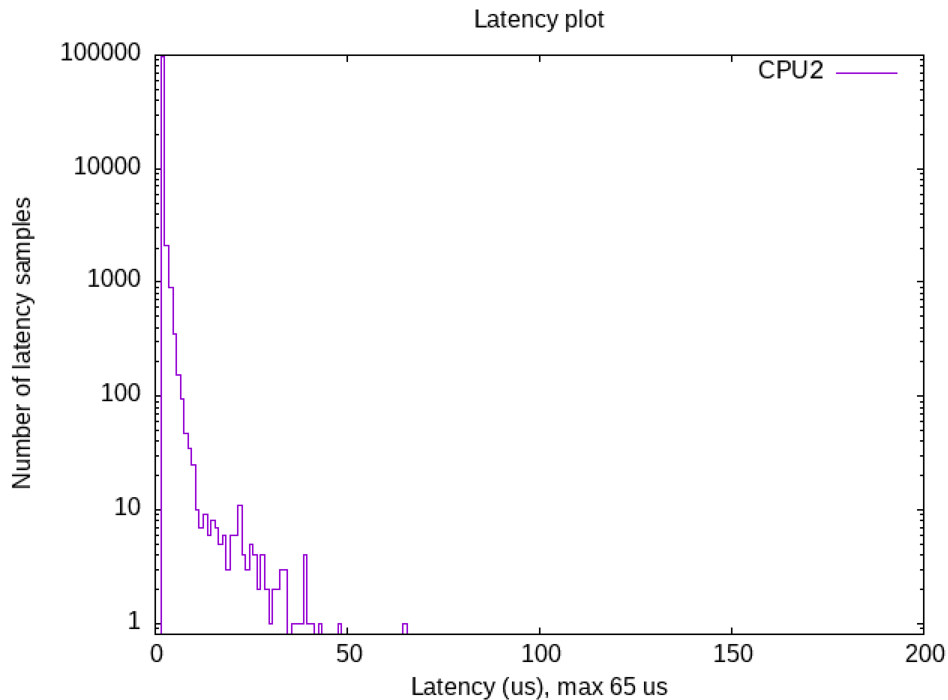

Figura 4.4: Hackbench con T separados.

4.2.1.3 Pruebas con CPUSET

Veamos ahora las diferencias que podemos encontrar utilizando CPUSET en vez de Taskset.

En el primer caso, se observa una distribución a lo largo del eje X de la prueba de estrés (Figura 4.6). En este caso, se ha ejecutado la prueba de estrés aislado en el núcleo 3 pero la generación de la gráfica ha sido ejecutada de forma nominal en la terminal de shell.

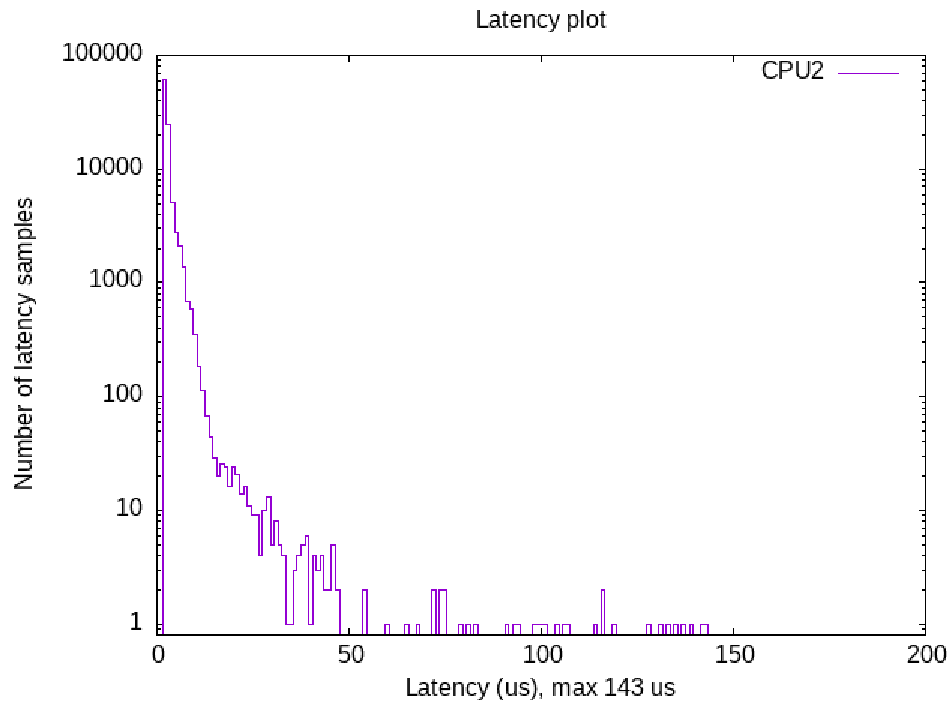

Figura 4.5: Hackbench on CPU2 and graphic on all CPUs.

Para el segundo caso, lo que se realiza es la ejecución de la prueba de estrés, así como la generación de la gráfica en el núcleo 3 de nuestra CPU. Como se observa en la Figura 4.5 ha habido una compactación muy elevada lo que esto indica que al ejecutar el gráfico en el mismo núcleo que la prueba de estrés, apenas interfiere un proceso con el otro.

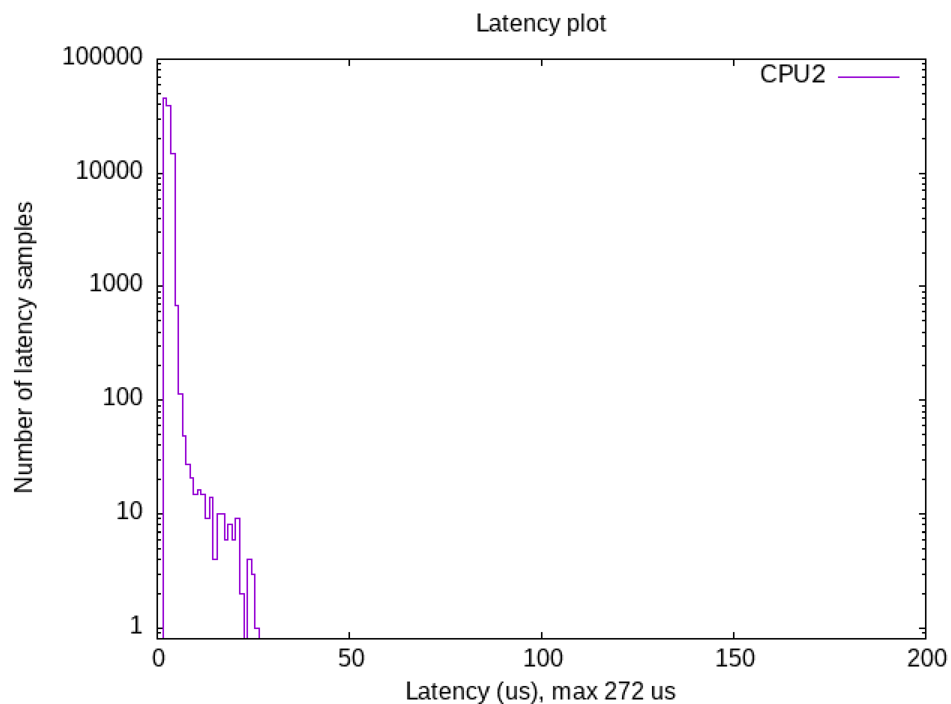

Figura 4.6: Hackbench and graphic on CPU2.

Aunque si observamos el siguiente caso (Figura 4.7), podemos obtener una mayor compactación aun, lo que nos lleva a tener un mayor rendimiento y menores interrupciones, si ejecutamos en núcleos aislados e independientes la prueba de estrés como la generación de la gráfica. Para este ejemplo, lo que se ha realizado es la ejecución de la prueba de hackbench en el núcleo 3 mientras que para la generación de la gráfica se ha usado el núcleo 4 (ambos aislados, como se ha comentado anteriormente).

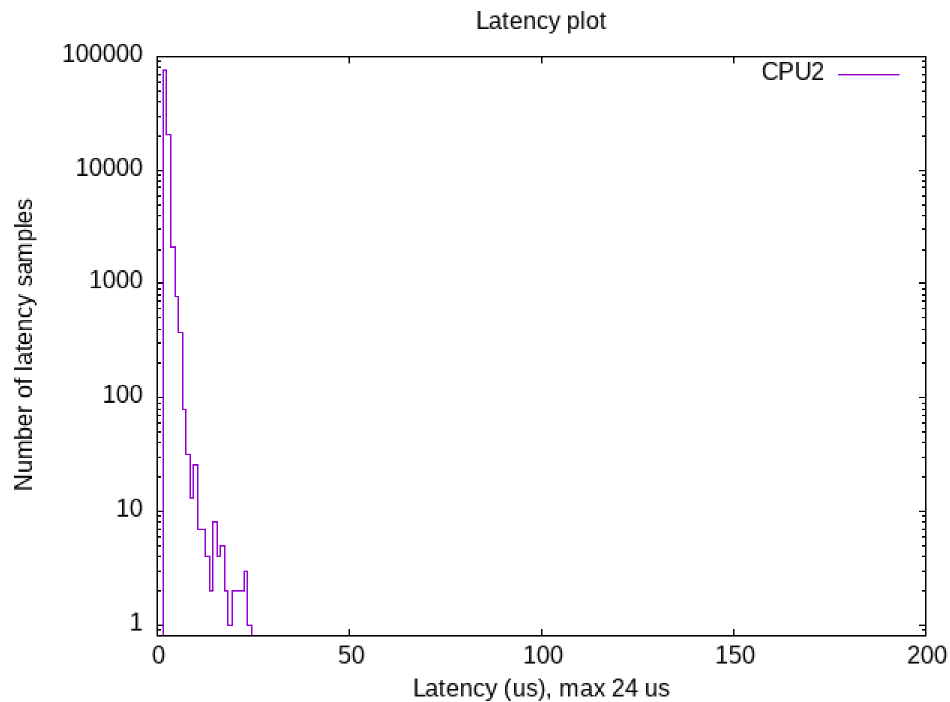

Figura 4.7: Hackbench on CPU2y gráficos en CPU3.

4.2.1.4 Stress

Ahora se analizan los datos obtenidos ejecutando la prueba de estrés denominada “Stress”.

Como en la prueba anterior, lo primero que se registra es la ejecución de la prueba en nominal.

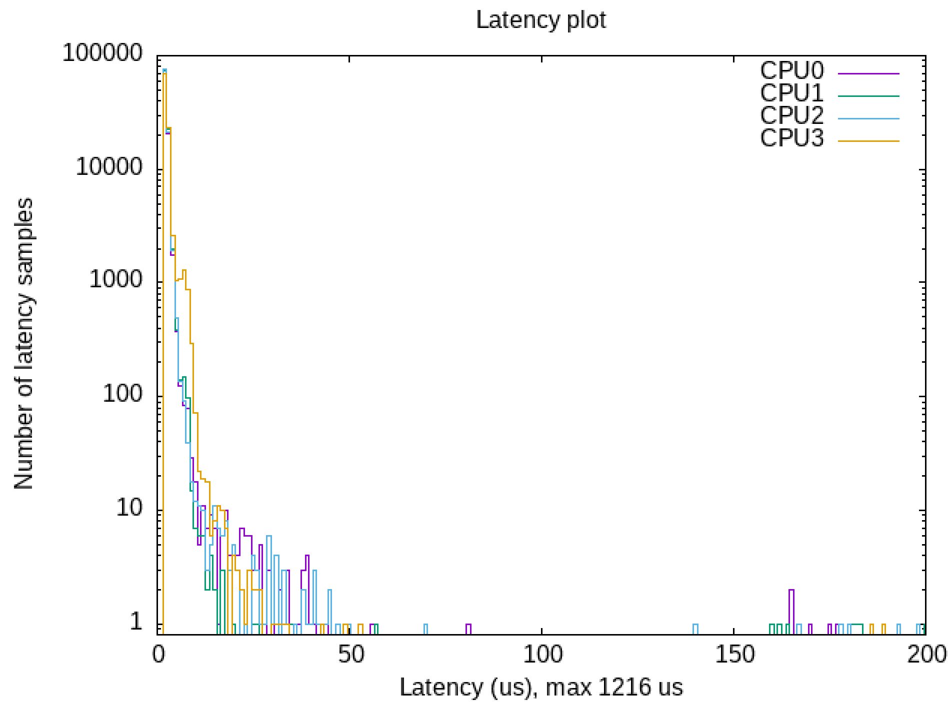

Figura 4.8: Stress simple.

Se puede apreciar en la figura anterior (Figura 4.8) que esta prueba es más exigente dado que se ve que la utilización de la gran mayoría de núcleos y por tiempos muy cortos, haciendo que haya muchas interrupciones, provocando también una mayor latencia.

4.2.1.5 Pruebas con Taskset

Se pasa a explicar los datos obtenidos en las pruebas que se han realizado utilizando taskset. Durante la primera prueba, se puede apreciar cierta mejora, pero aun así no se nota demasiada mejoría como se aprecia en la siguiente Figura 4.9.

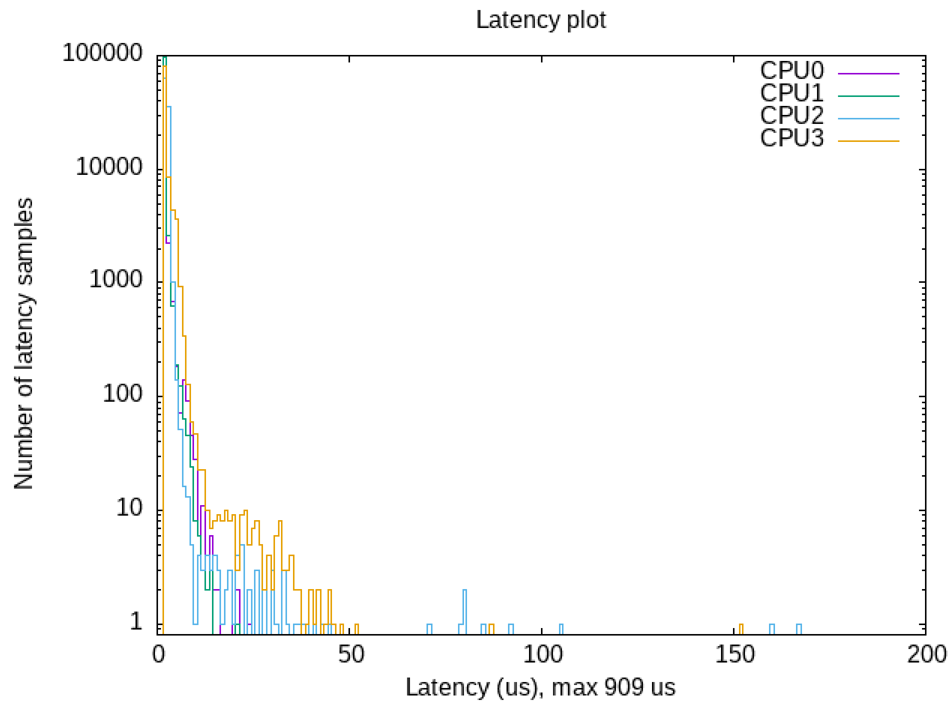

Figura 4.9: Stress con T simple.

A continuación, se realizan las pruebas aislando en el mismo núcleo la prueba de estrés y la generación de la gráfica (Figura 4.10). Se observa que se interrumpen entre ellos, pero no se generan grandes picos de latencia y se concentra el trabajo para realizarlo lo antes posible.

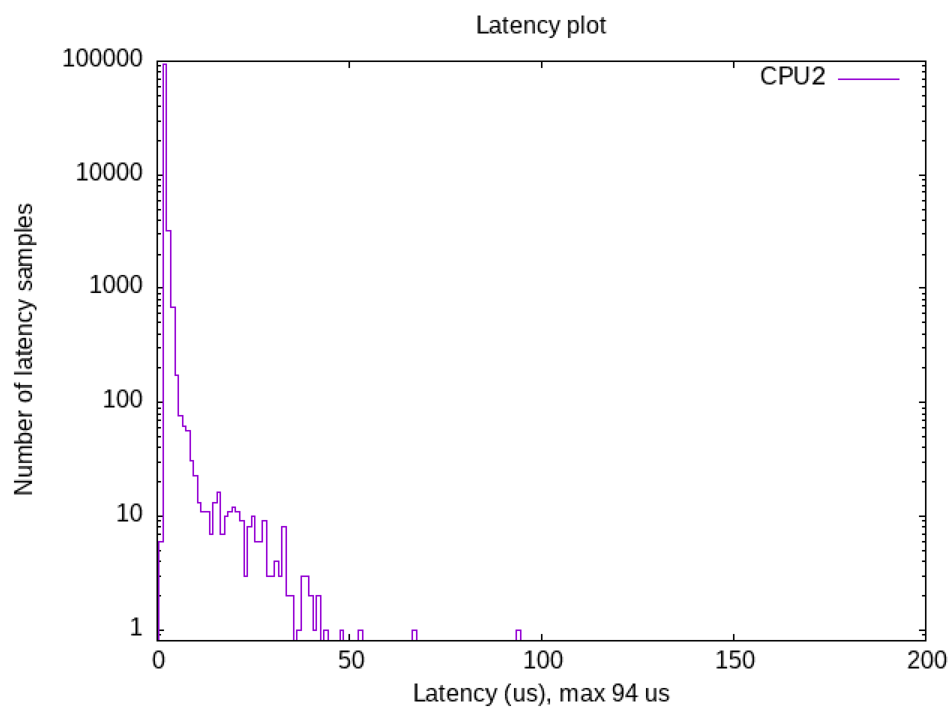

Figura 4.10: Stress con T juntos.

Sin embargo, si tenemos en cuenta separar los dos procesos en dos núcleos distintos, podremos notar mucha más mejoría (Figura 4.11). Como se aprecia, todo el trabajo se ha concentrado indicando que apenas hay latencia, y que no se interrumpen entre los dos procesos que se están ejecutando.

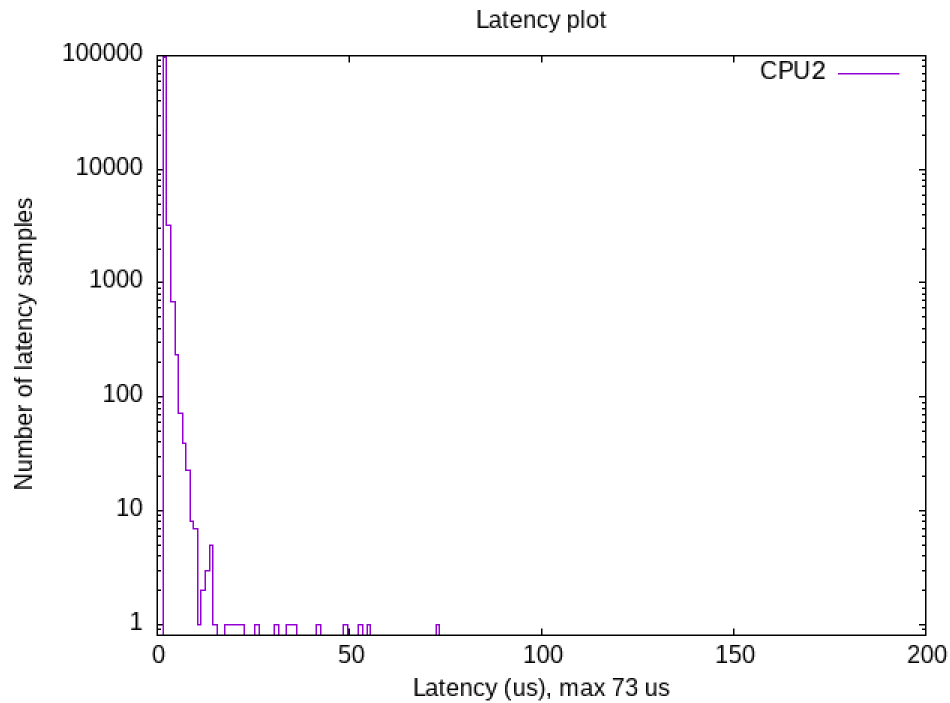

Figura 4.11: Stress con T separados.

4.2.1.6 Pruebas con CPUSET

A continuación, se ejecutan los dos procesos con el método de aislamiento de CPUSET. Como se ve en cada caso, se obtienen mejores datos que en el resto de los métodos de aislamiento.

Se pasa a analizar el primer caso, el cual consiste en ejecutar de forma nominal (en una shell sin aislamiento) la generación del gráfico mientras que, por otro lado, se ejecuta en el núcleo 3 el cual tiene el aislamiento basado en CPUSET la prueba de estrés. Lo resultados se muestran en la siguiente Figura 4.12.

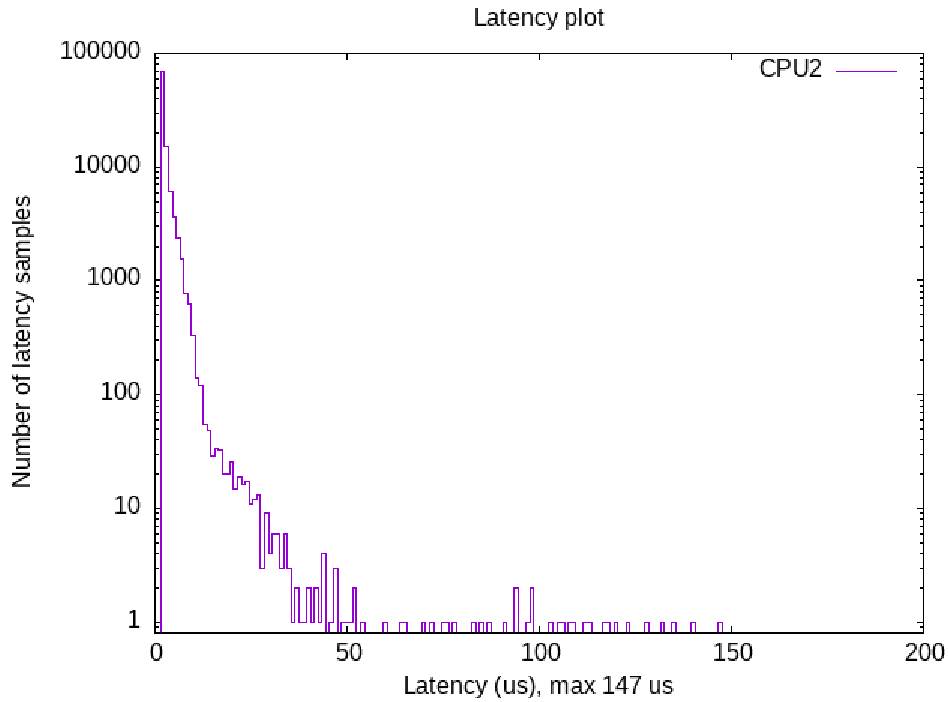

Figura 4.12: Stress en CPU2 y gráficos en todas las CPUs.

Se pasa al siguiente caso de prueba. Esta vez, lo que se hace será aislar los dos procesos en el mismo núcleo (Figura 4.13).

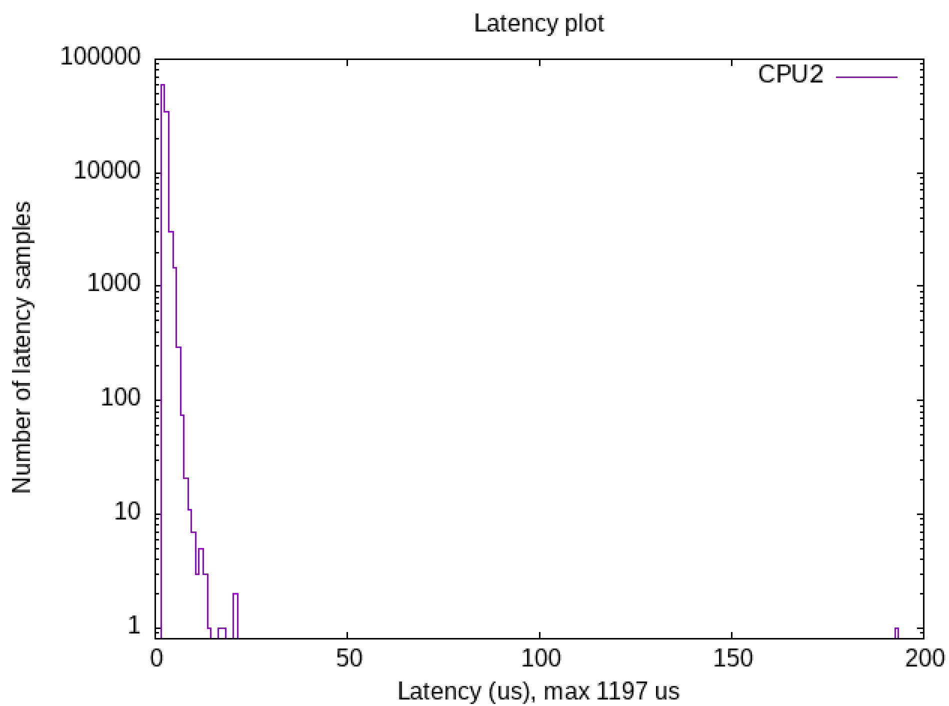

Figura 4.13: Stress y gráficos en CPU2.

Como se aprecia, se ha obtenido una compresión de la latencia lo que indica que la tarea que se estaba ejecutando no ha sufrido apenas interrupciones. Por otro lado, es cierto, que se observa una latencia máxima de 1197 μs, pero esto es debido a algún proceso ajeno a la ejecución del propio programa, como

por ejemplo, mover el ratón en ese momento, algún proceso en background, alguna interrupción por E/S,. . .

Por último, se ejecutan los dos procesos en núcleos separados. El proceso de generación de la gráfica se ejecuta en el núcleo 4 mientras que el proceso de estrés se ejecuta en el núcleo 3. Se puede diferenciar, que hay una alta carga de trabajo al principio indicando que los procesos no se interrumpen entre ellos e intentan ejecutarse lo más rápido posible (Figura 4.14).

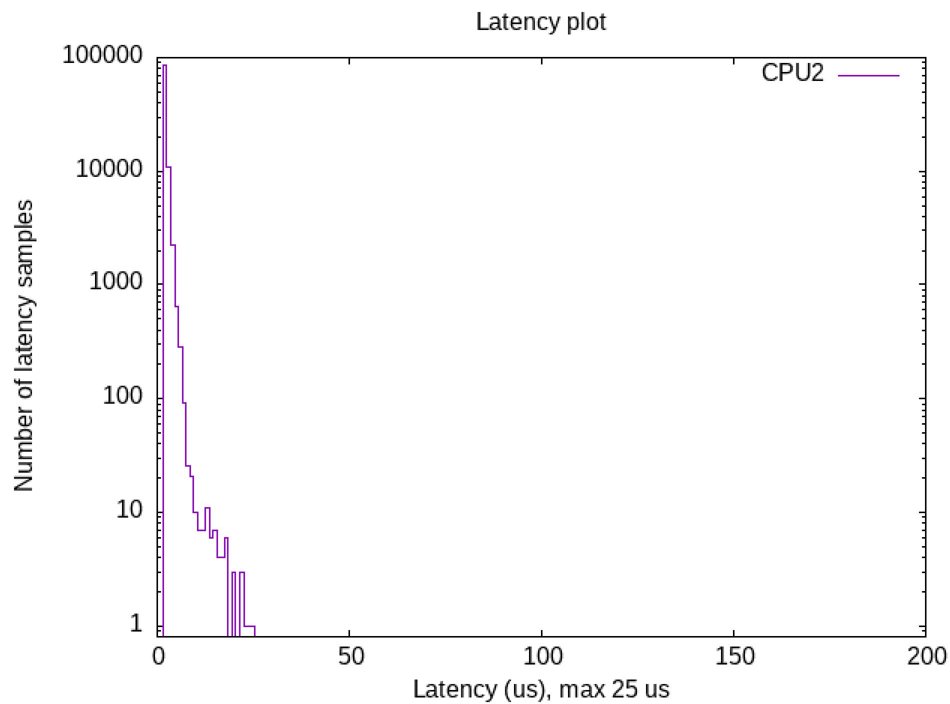

Figura 4.14: Stress en CPU2 y gráficos en CPU3.

4.2.1.7 Sysbench

Se pasa a la última prueba de estrés que realizaremos en Ubuntu. En esta ocasión, se analizan los datos obtenidos de la prueba de estrés llamada “Sysbench”.

Se empieza ejecutando la prueba en la terminal de forma normal, es decir, sin ningún mecanismo de aislamiento.

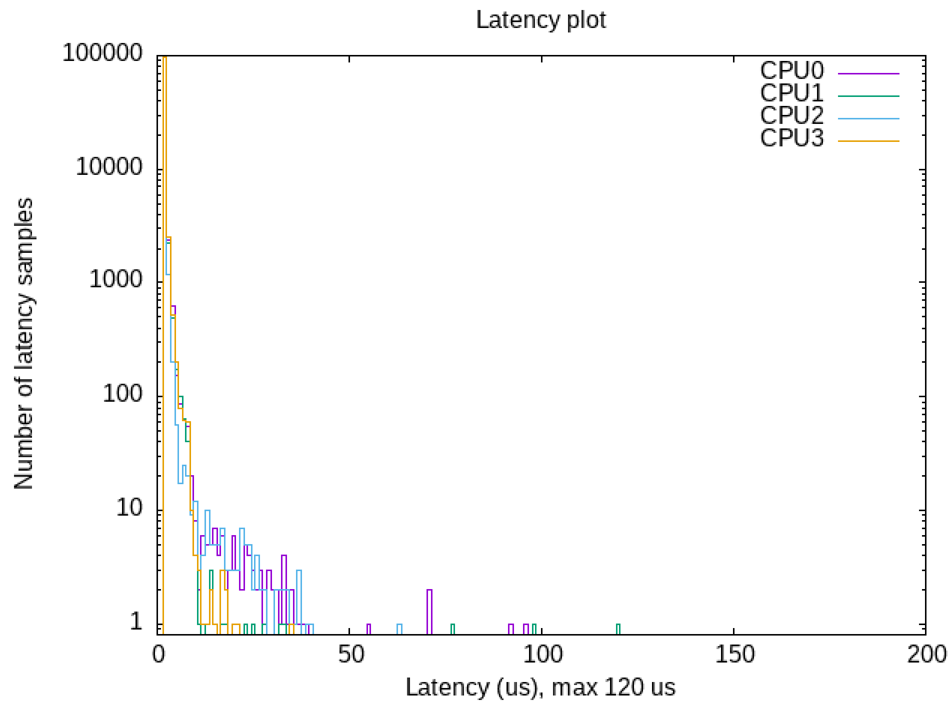

Figura 4.15: Sysbench simple.

Como se aprecia en la Figura 4.15, la ejecución de la prueba de estrés junto con la generación de los gráficos, así como el resto de las tareas que este ejecutando el sistema, hacen que el resultado no sea demasiado óptimo. Se va a ver a continuación como mediante los distintos mecanismos de aislamiento se pueden obtener mejores resultados.

4.2.1.8 Pruebas con Taskset

El primer mecanismo a analizar es taskset, mediante el script "Latencygraph.sh" (que se puede observar en el Anexo A). El primer análisis que haremos será aislando únicamente la prueba de estrés, mientras que la generación del gráfico se sigue ejecutando junto con el resto de las tareas del sistema (Figura 4.16).

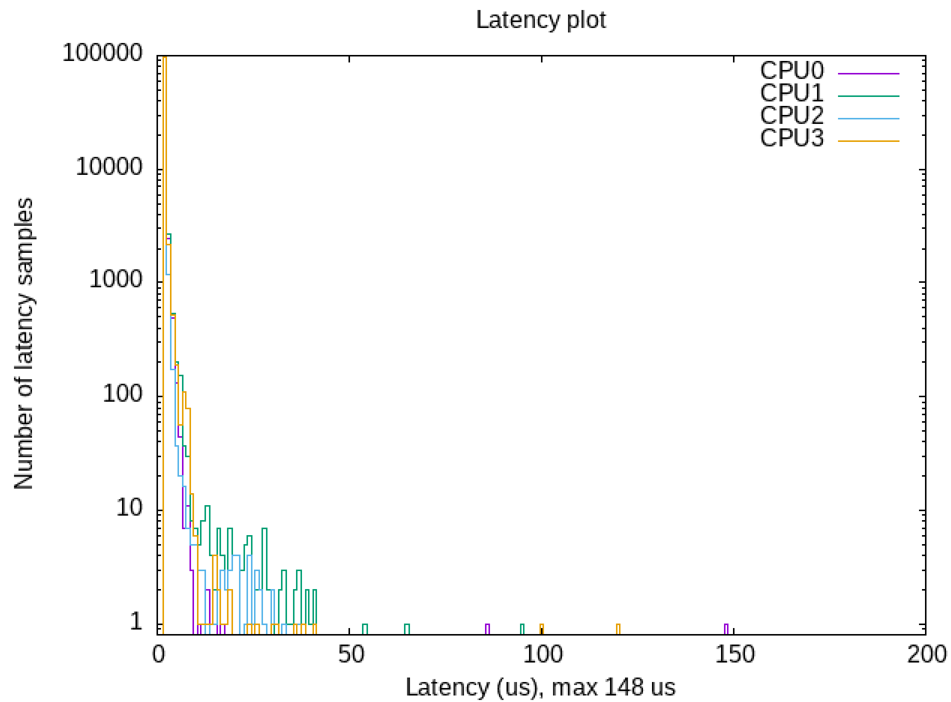

Figura 4.16: Sysbench con T simple.

Si se evalúan los datos obtenidos, se puede apreciar una menor latencia global, lo que quiere decir que hay menos interrupciones que cuando se ejecuta la prueba de estrés junto con el resto de los procesos del sistema. A continuación, se pasa a ejecutar la generación de las gráficas junto con el proceso de estrés en el mismo núcleo, aislados del resto de procesos del sistema.

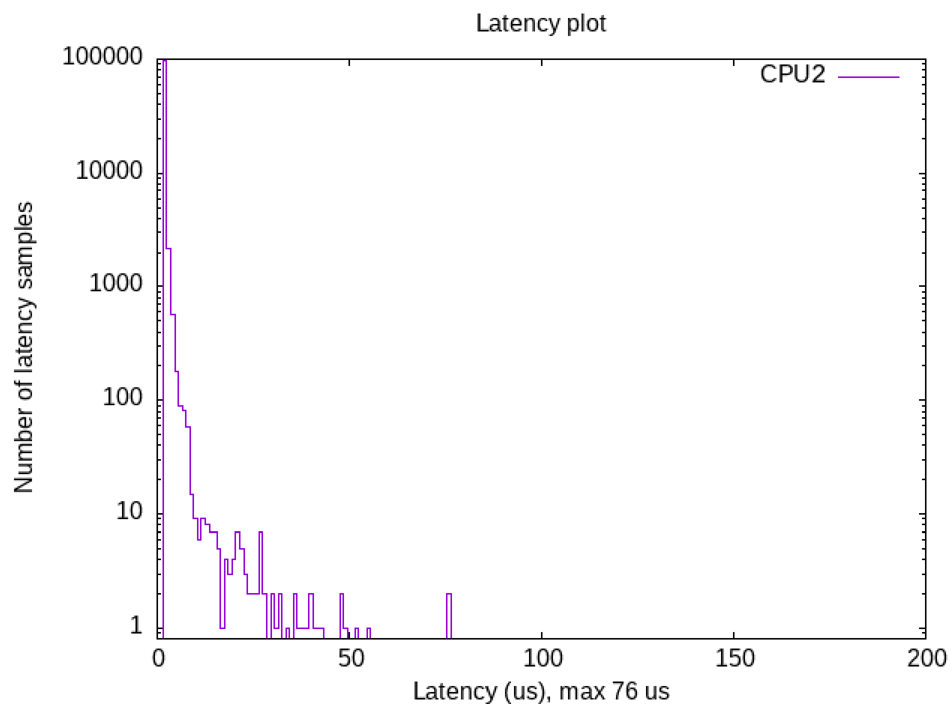

Figura 4.17: Sysbench con T juntos.

Como se ha detectado en otras ocasiones, analizando los datos que se han obtenido (Figura 4.17), se puede observar una menor latencia que en las dos pruebas anteriores, lo que indica que sí que es efectiva esta estrategia. Aun así, se analizará que es lo que sucede cuando se separa la prueba de estrés en un núcleo y la generación de la gráfica en otro núcleo.

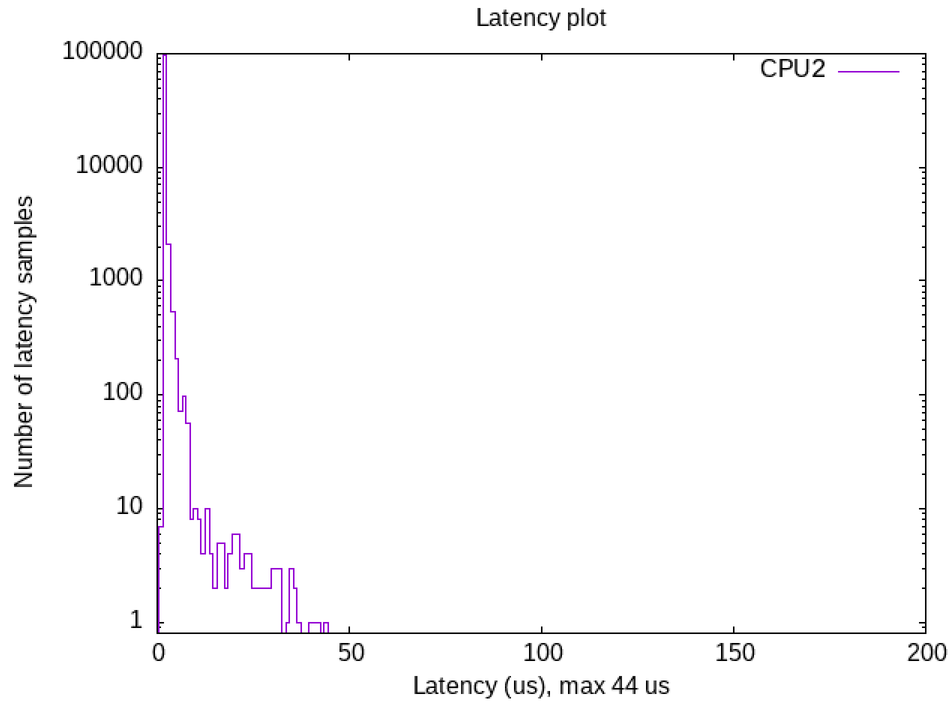

Figura 4.18: Sysbench con T separados.

Se contempla una latencia mucho menor que en todos los casos anteriores (Figura 4.18), con lo que lo que se haría sería aislar en un núcleo el programa el cuál se quiera obtener un mayor rendimiento y que no sea interrumpido por ningún otro proceso.

4.2.1.9 Pruebas con CPUSET

Se pasan a evaluar los datos obtenidos mediante el mecanismo de CPUSET.

En primera instancia lo que se hará será lo mismo que en el anterior mecanismo, es decir, se procede a aislar la prueba de estrés en un núcleo y la generación de los gráficos junto con el resto de los procesos del sistema.

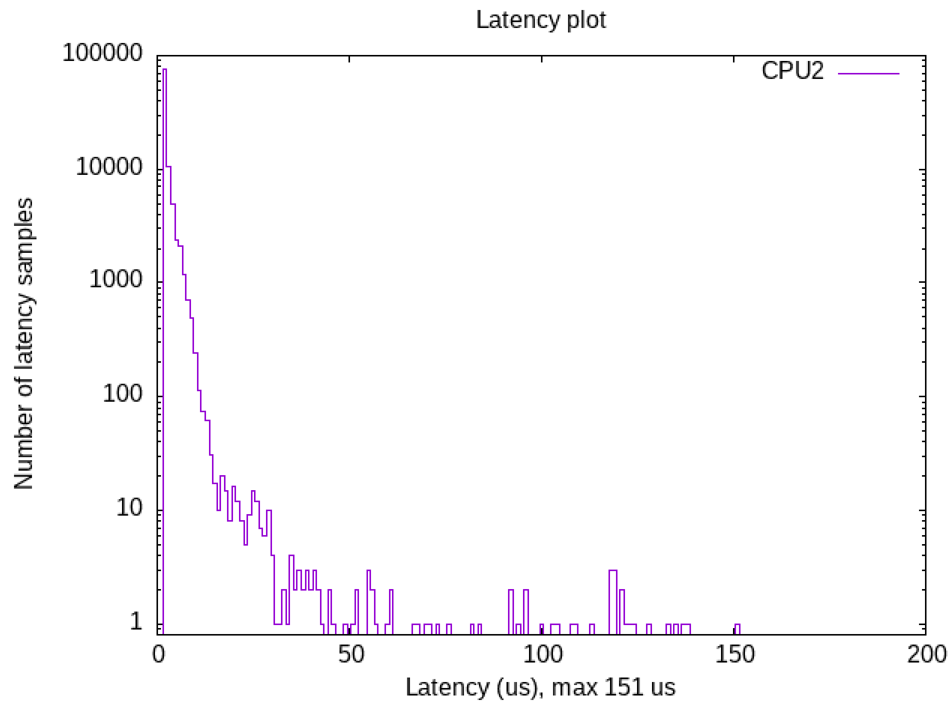

Figura 4.19: Sysbench en CPU2 y gráficos en todas las CPUs.

Si se examinan los datos obtenidos (Figura 4.19), no se ve mucha mejoría bajo las mismas condiciones que en el mecanismo de taskset, pero a continuación se va a proceder a analizar los datos que se obtienen con el siguiente escenario (Figura 4.20). En este escenario, lo que se va a hacer es bajo el mismo núcleo se va a ejecutar tanto la prueba de estrés, así como la generación de la gráfica para observar la latencia.

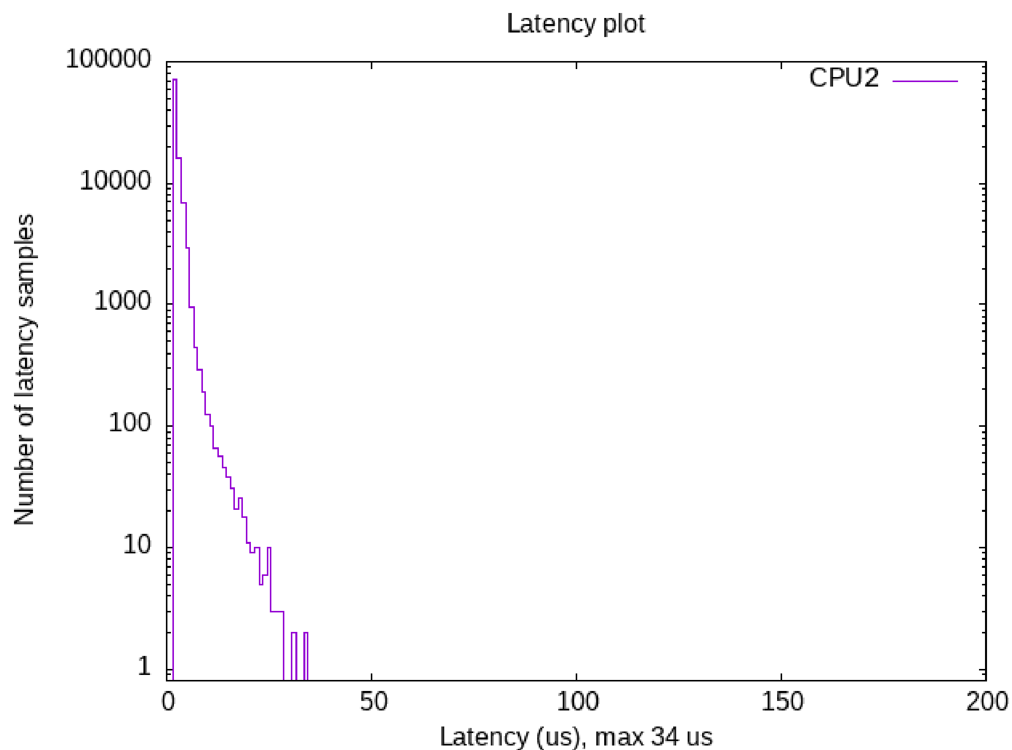

Figura 4.20: Sysbench y gráficos en CPU2.

En unas condiciones similares a las producidas en el escenario de taskset, es decir, aislando los dos procesos bajo el mismo núcleo, se puede observar que hay una mejor latencia y que los procesos apenas se interrumpen entre sí (Figura 4.20). Por último, se va a contemplar el escenario en el que se ejecuta en un núcleo la prueba de estrés y en otro núcleo distinto la generación de la gráfica.

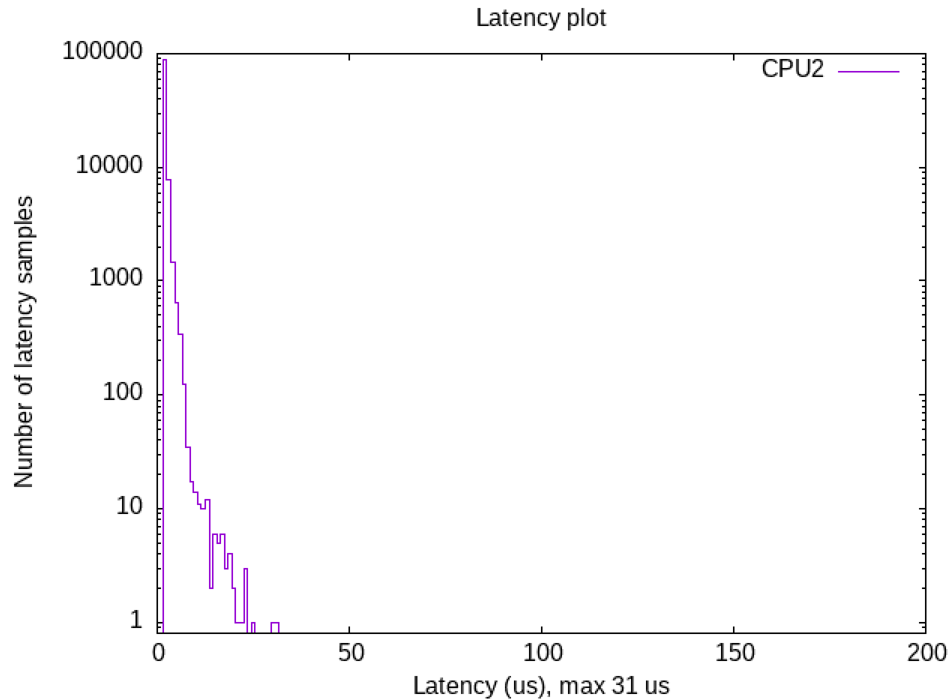

Figura 4.21: Sysbench en CPU2 y gráficos en CPU3.

Se aprecia que los datos obtenidos son mejores en cuanto a la latencia general (Figura 4.21), ya que no hay interrupciones ni entre los propios procesos, ni con los procesos propios del sistema.

4.2.2 Pruebas con la Raspberry

En esta sección se van a analizar los datos obtenidos de la ejecución del programa para Raspberry descrito en la sección 3.4. Este programa genera una señal cuadrada de 5 Mhz de frecuencia y se pretende analizar la estabilidad temporal de esta señal dependiendo del tipo de configuración elegida para su ejecución.

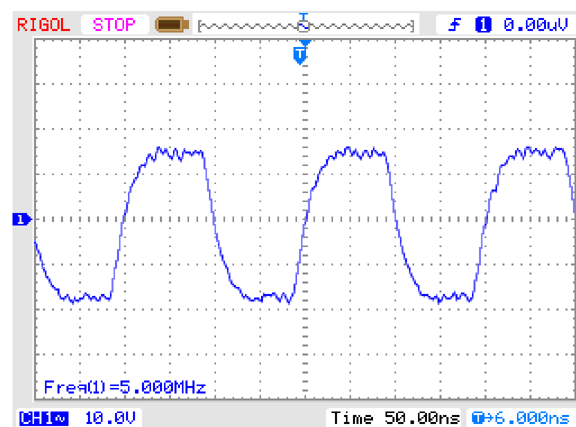

Figura 4.22: Imagen obtenida del osciloscopio con un Time de 50 ns.

La figura 4.22 muestra la señal generada con una escala de tiempo de 50 ns por división. De forma general la señal deberá tener de forma periódica 100 ns en alto y 100 ns en bajo. Para analizar la estabilidad de estos tiempos se ha ejecutado el programa en dos configuraciones:

- Nominal: Sin aislamiento, compartiendo el núcleo con los demás procesos.
- Taskset. Ejecución del programa aislado en un núcleo mediante taskset.

Para cada ejecución se ha capturado un segmento de un segundo de la señal de salida mediante un analizador lógico para su posterior análisis.

4.2.2.1 Ejecución nominal

Lo primero que se ha realizado es la ejecución de la prueba de estrés de forma nominal, es decir, sin ningún tipo de aislamiento y con la carga habitual del SO Raspbian.

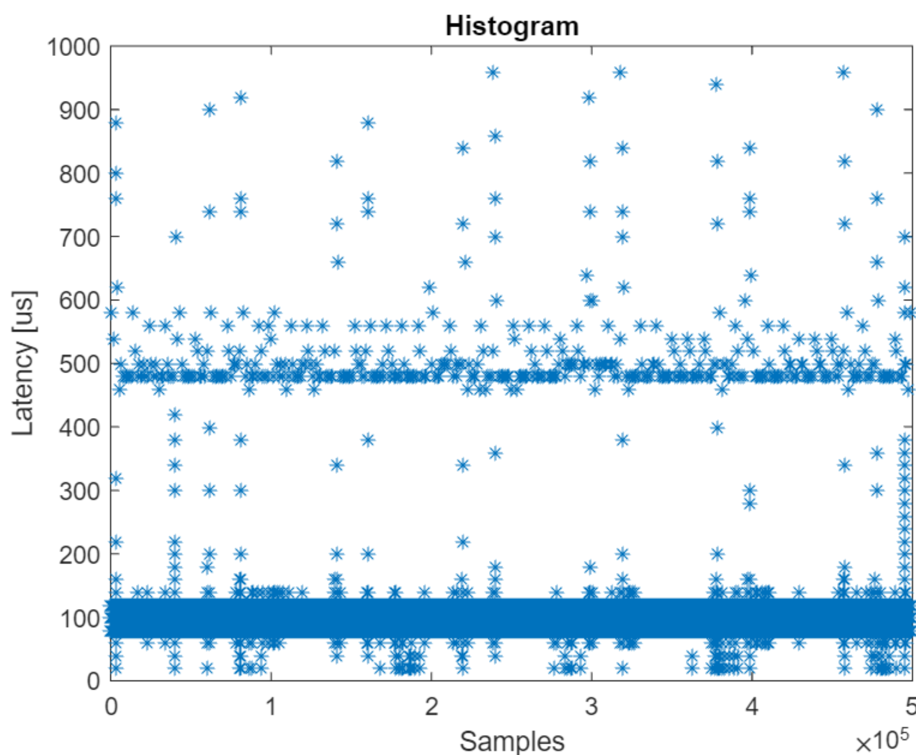

Figura 4.23: Datos de la ejecución nominal en Raspberry.

Como se puede observar en la Figura 4.23, la distribución de los puntos presenta una gran variabilidad, aunque se puede apreciar que la mayoría de los puntos se distribuyen en un valor entorno a los 100 ns. Lo que nos quiere decir esta representación, es que la onda que se está generando por parte de la Raspberry, está sufriendo interrupciones y desalojos por parte del planificador que hacen que el periodo generado no sea constante en torno 100 ns sino que hay una gran dispersión y esto genere una señal con poca precisión.

La tabla 4.1 muestra los valores concretos de la media y desviación típica del segmento de tiempo capturado. Se observa que el valor medio está próximo al valor deseado pero la dispersión es muy alta.

Media	101, 5 ns
Desviación	139, 5 ns

Tabla 4.1: Media y desviación típica nominal

4.2.2.2 Ejecución mediante Taskset en un CPUSET aislado

A continuación, se procede a aislar un núcleo de la Raspberry para ejecutar el programa de generación de señales de forma que ocupe en exclusiva un núcleo y que no se vea interrumpida por ningún otro proceso. Para ello se han seguido los siguientes pasos:

En primer lugar, se procederá a la creación de los CPUSET `housekeeping` y `isolated`. El primero, formado por los núcleos 0,1 y 2 y al que se trasladarán todos los procesos del sistema dejando libre el núcleo 3. Este proceso está descrito en la figura 4.24.

```
cd /sys/fs/cgroup/cpuset
mkdir housekeeping
mkdir isolated
echo 0-2 > housekeeping/cpuset.cpus         # Mueve los núcleos 0-2 al grupo housekeeping
echo 3 > isolated/cpuset.cpus               # Mueve el núcleo 3 al grupo isolated
echo 0 > isolated/cpuset.sched_load_balance # Deshabilita el balanceo de procesos en el cpuset isolated
while read P
do
  echo $P > housekeeping/cgroup.procs       # Mueve todos los PID al grupo housekeeping
done < cgroup.procs                         # Todos los PIDs
```

Figura 4.24: Creación de los CPUSET.

A continuación, se define la afinidad de las interrupciones para que todas sean procesadas en los núcleos del grupo `housekeeping`. excluyendo al núcleo 3, figura 4.25.

```
# Mueve Irqs a los Núcleos0-2 (excluido Núcleo 3)
for I in $(ls /proc/irq)
do
    if [ -d "/proc/irq/$I" ]
    then
        echo "Affining vector $I to CPUs 0-2"
        echo 0-2 > /proc/irq/$I/smp_affinity_list
    fi
done
```

Figura 4.25: Definición de la afinidad de las interrupciones.

Finalmente se lanzará el programa de prueba `signal` en el núcleo 3 mediante la orden `taskset`. Se define la captura de eventos del planificador para ver cuáles son las perturbaciones que pueda sufrir el programa, si es que existen, figura 4.26.

```
TRACING=/sys/kernel/debug/tracing/
# Deshabilitar el rastreo de eventos
echo 0 > $TRACING/tracing_on
# Limpiar trazas anteriores
echo > $TRACING/trace
# Habilitar eventos de planificación
echo 1 > $TRACING/events/sched/sched_switch/enable
# Habilitar eventos de interrupción
echo 1 > $TRACING/events/irq_vectors/enable
# Iniciar rastreo
echo 1 > $TRACING/tracing_on
# Ejecutar el programa "signal" durante 5 segundos en el núcleo 3
taskset -c 3 ./signal &
USER_LOOP_PID=$!
sleep 5
kill $USER_LOOP_PID
# Deshabilitar rastreo y guardar las trazas del núcelo 3
echo 0 > $TRACING/tracing_on
cat $TRACING/per_cpu/cpu3/trace > trace.3
```

Figura 4.26: Definición de los eventos de planificación y ejecución del programa de prueba.

Los resultados de la ejecución del programa se muestran a continuación.

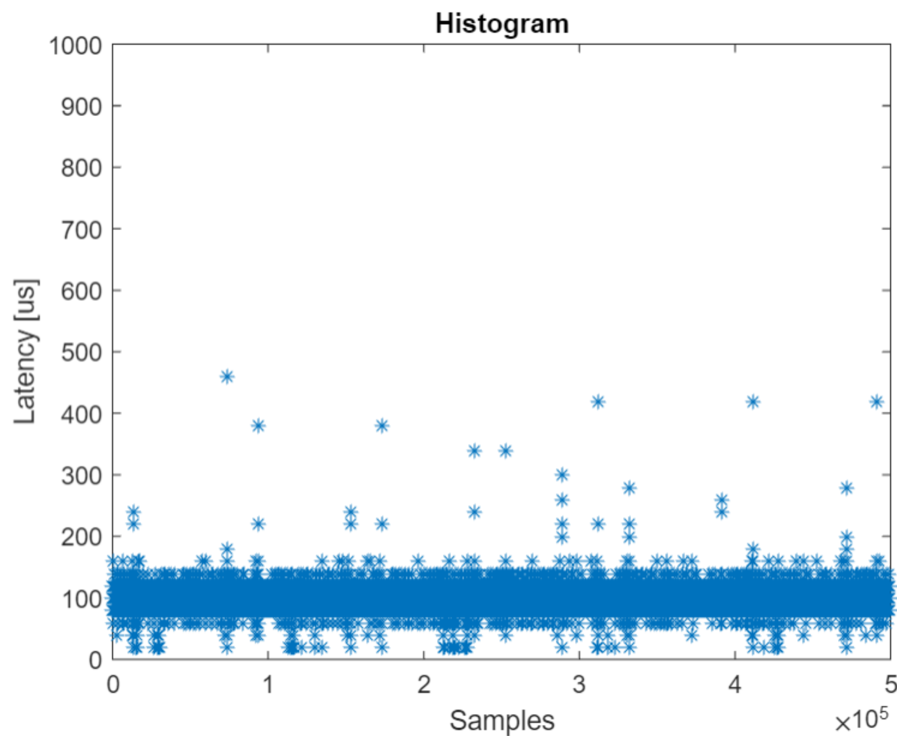

Figura 4.27: Datos de la ejecución con Taskset en Raspberry.

Ahora se puede apreciar que en la Figura 4.27, los puntos están más agrupados en torno al valor esperado que en la Figura 4.23, y que no se existe tanta dispersión de los tiempos alrededor del valor esperado.

Media	$100,38$ ns
Desviación	$5,69$ ns

Tabla 4.2: Media y desviación típica usando taskset

A pesar de que los datos implican que el programa se ejecuta en exclusiva la mayor parte el tiempo, todavía existen algunas interferencias. Estas interferencias están provocadas por la interrupción del timer. Esta interrupción es procesada por parte de cada núcleo y de forma general no puede ser deshabilitada. Es preciso recompilar el kernel con la opción “CONFIG_HZ_PERIODIC=y” y usar el parámetro de arranque del kernel para indicar a que núcleos se aplica. Por ejemplo: `nohz_full=3` indicaría que el núcleo 3 no debe recibir la interrupción del timer.

Capítulo 5

Conclusiones y líneas futuras

A lo largo de este trabajo de fin de grado se han ido exponiendo los distintos mecanismos junto con sus resultados del aislamiento de un núcleo del resto de carga de procesamiento del procesador. De entre todos los mecanismos de aislamiento que se han probado, se podría destacar que el mecanismo que pudiera ser más efectivo sería con el mecanismo de aislamiento de CPUSET ya que durante las pruebas que se han realizado ha sido el que menos latencia ha obtenido en las pruebas de estrés.

Este trabajo muestra que en un entorno con kernel Linux genérico, se ha conseguido reducir los tiempos de latencia en ocasiones hasta 200 μs con lo que con lleva que el programa en tiempo real que se ejecutase pudiera tener una mejor respuesta y también menores interrupciones del sistema.

Hay demonios (kernel threads) o programas que se lanzan por el propio kernel de Linux que no permiten un aislamiento íntegro del sistema, con lo que deberíamos que tener esto en cuenta en una posible planificación de tareas. Además, están las interrupciones del timer que solo pueden ser deshabilitadas del todo en cada núcleo mediante una recompilación del kernel para habilitar la opción de arranque `nohz_full`.

Así pues, una línea futura de investigación podría ser ejecutar estas pruebas de estrés y su análisis correspondiente de los datos, pero una versión del kernel de Linux totalmente configurada para sistemas en tiempo real.

En el caso de la generación de señales externas se ha comprobado su viabilidad y podría plantearse usar un núcleo para simular por software controladores de comunicaciones como podría ser una interfaz serie tipo SPI, funcionando a 5 Mhz como frecuencia de reloj.

También se podría realizar el mismo estudio, pero con el sistema operativo de Windows, ya que como se vio en la introducción dispone de ciertas herramientas que pueden proporcionar cierto aislamiento de una aplicación en un núcleo. Se tendría que comprobar si estas técnicas son eficientes y suficientes para según que aplicaciones en tiempo real se quieran desarrollar en Windows. Además de comparar los resultados que se han obtenido en este estudio sobre Linux.

Bibliografía

[1] ARM Limited, *ARM Architecture Reference Manual for A-profile architecture*, ARM Limited, Cambridge, 2022.

[2] B. Moyer, "Chapter 1 - introduction and roadmap," in *Real World Multicore Embedded Systems*, B. Moyer, Ed. Oxford: Newnes, 2013, pp. 1–10. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780124160187000018

[3] W. J. Chan, A. B. Kahng, S. Nath, and I. Yamamoto, "The ITRS MPU and SOC system drivers: Calibration and implications for design-based equivalent scaling in the roadmap," in *2014 IEEE 32nd International Conference on Computer Design (ICCD)*, Oct 2014, pp. 153–160.

[4] J. A. Stankovic and K. Ramamritham, *Hard Real-Time Systems.* Washington, DC, USA: IEEE Computer Society Press, 1988.

[5] "Linux foundation. the real-time linux." https://wiki.linuxfoundation.org/realtime/start [Último acceso 15/mayo/2022].

[6] F. REGHENZANI, G. MASSARI, W. Fornaciari, and P. Milano, "The real-time linux kernel: A survey on preempt _ rt the real-time linux kernel: A survey on preempt _ rt," *no. February*, 2019.

[7] S. Han, "System Design for Software Packet Processing," Ph.D. dissertation, EECS Department, University of California, Berkeley, 2019.

[8] M. Darianian, C. Williamson, and I. Haque, "Experimental evaluation of two openflow controllers," in *2017 IEEE 25th International Conference on Network Protocols (ICNP)*, 2017, pp. 1–6.

[9] K. Okamoto, "Core partitioning technique on multicore linux systems," Tech. Rep., December 2007.

[10] L. M. Pages, *cpuset(7) Linux manual page*, November 2020.

[11] A. Tsariounov, *Shielding Linux Resources*, SUSE SLL, May 2022.

[12] *Supervisión y gestión del estado y el rendimiento del sistema.*, Red Hat, March 2021.

[13] "Ayuda del comando start para windows." https://docs.microsoft.com/es-es/windows-server/administration/windows-commands/start [Último acceso 02/febrero/2022].

[14] "Cómo lanzar un proceso con afinidad de cpu." https://docs.microsoft.com/es-es/archive/blogs/santhoshonline/how-to-launch-a-process-with-cpu-affinity-set [Último acceso 02/febrero/2022].

[15] "Utilización del comando process.processoraffinity." https://docs.microsoft.com/en-us/dotnet/api/system.diagnostics.process.processoraffinity?view=net-6.0 [Último acceso 02/febrero/2022].

[16] "Luajit," https://luajit.org/luajit.html [Último acceso 15/mayo/2022].

[17] “Generación de gráficas para la medición de la latencia.” https://www.osadl.org/Create-a-latency-plot-from-cyclictest-hi.bash-script-for-latency-plot.0.html [Último acceso 20/diciembre/2021].

[18] A. Guide, “Realview® compilation tools,” 2002.

[19] “Archivo smp.h de kernel linux v5.16.10,” https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/include/linux/smp.h?h=v5.16.10 [Último acceso 04/febrero/2022].

[20] Broadcom Europe Ltd., *BCM2711 ARM Peripherals*, Raspberry Pi Ltd, Cambridge, 2012.

[21] “Información sobre gnu/linux en wikipedia,” http://es.wikipedia.org/wiki/GNU/Linux [Último acceso 04/mayo/2021].

Apéndice A

Código fuente Latencygraph

```
#!/bin/bash

######################### VARIABLES #########################

RUTA="/home/rasalghul/TFG/graphic"
CYCL="cyclictest -l100000 -m -Sp90 -i200 -h400 -q"
HACK="hackbench -s 512 -l 1024 -P"
STRESS="stress --cpu 8 --io 4 --vm 2 --vm-bytes 128M --timeout 10s"
SYSB="sysbench cpu --cpu-max-prime=50000 run"
NAME=""
flag=0

################################################################

######################### FUNCTIONS #########################

cyclic_opt(){
  if [ "$1" == 1 ]
  then
     taskset -c 2 $CYCL >$RUTA/output
     NAME="${NAME}-tg"
  elif [ "$1" == 2 ]
  then
     taskset -c 3 $CYCL >$RUTA/output
     NAME="${NAME}-sp"
  else
     $CYCL >$RUTA/output
     NAME="${NAME}-s"
  fi
}

################################################################

# 0. Choice the test would be execute.

case "$1" in
```

```
37      -h) case "$2" in
38          -c) taskset -c 2 $HACK >> /dev/null 2>&1 &
39            NAME="T-hack"
40            ;;
41
42          *) $HACK >> /dev/null 2>&1 &
43             NAME="Hack"
44             ;;
45        esac;;
46      -t) case "$2" in
47          -c) taskset -c 2 $STRESS >> /dev/null 2>&1 &
48            NAME="T-str"
49            ;;
50          *) $STRESS >> /dev/null 2>&1 &
51            NAME="Str"
52            ;;
53        esac;;
54      -y) case "$2" in
55          -c) taskset -c 2 $SYSB >> /dev/null 2>&1 &
56            NAME="T-sys"
57            ;;
58          *) $SYSB >> /dev/null 2>&1 &
59            NAME="Sys"
60            ;;
61        esac;;
62    esac
63
64    # 1. Execution of Cyclictest
65    cyclic_opt $3
66
67    # 2. Get maximum latency
68    max=`grep "Max Latencies" $RUTA/output | tr " " "\n" | sort -n | tail -1 | sed
          s/^0*//`
69
70    # 3. Grep data lines, remove empty lines and create a common field separator
71    grep -v -e "^#" -e "^$" $RUTA/output | tr " " "\t" >$RUTA/histogram
72
73    # 4. Set the number of cores
74    cores=`nproc`
75
76    # 5. Create two-column data sets with latency classes and frequency values for
          each core, for example
77    for i in `seq 1 $cores`
78    do
79      column=`expr $i + 1`
80      cut -f1,$column $RUTA/histogram >$RUTA/histogram$i
81    done
82
83    # 6. Create plot command header
84    echo -n -e "set title \"Latency plot\"\n\
85    set terminal png\n\
```

```
86  set xlabel \"Latency (us), max $max us\"\n\
87  set logscale y\n\
88  set xrange [0:*]\n\
89  set yrange [0.8:*]\n\
90  set ylabel \"Number of latency samples\"\n\
91  set output \"$RUTA/$NAME.png\"\n\
92  plot " >$RUTA/plotcmd
93
94  # 7. Append plot command data references
95  for i in `seq 1 $cores`
96  do
97    if test $i != 1
98    then
99      echo -n ", " >>$RUTA/plotcmd
100   fi
101   cpuno=`expr $i - 1`
102   if test $cpuno -lt 10
103   then
104     title=" CPU$cpuno"
105    else
106     title="CPU$cpuno"
107   fi
108   echo -n "\"$RUTA/histogram$i\" using 1:2 title \"$title\" with histeps" >>
        $RUTA/plotcmd
109 done
110
111 # 8. Execute plot command
112 gnuplot -p $RUTA/plotcmd
113
114 find $RUTA -type f -not -name '*.png' -delete
```

Apéndice B

Códigos fuentes usados en la Raspberry

En este apéndice se muestran completos los distintos códigos que se han usado con la Raspberry.

B.1 enable_ccr.c

```
#include <linux/module.h>
#include <linux/kernel.h>

void enable_ccr(void *info) {
  // Set the User Enable register, bit 0
  asm volatile ("mcr p15, 0, %0, c9, c14, 0" :: "r" (1));

  // Enable all counters in the PNMC control-register
  asm volatile ("MCR p15, 0, %0, c9, c12, 0\t\n" :: "r"(1));
  //asm volatile ("MCR p15, 0, %0, c9, c12, 0\t\n" :: "r"(8)); // divide 64

  // Enable cycle counter specifically
  // bit 31: enable cycle counter
  // bits 0-3: enable performance counters 0-3
  asm volatile ("MCR p15, 0, %0, c9, c12, 1\t\n" :: "r"(0x80000000));

// Overflow Flag Status Register
// asm volatile ("MCR p15, 0, %0, c9, c12, 3\t\n" :: "r"(0x8000000f));

// Clear overflows
//asm volatile ("MCR p15, 0, %0, c9, c12, 3\t\n" :: "r"(0x8000000f));
}

int init_module(void) {
  // Each cpu has its own set of registers
  on_each_cpu(enable_ccr,NULL,0);
  printk (KERN_INFO "Userspace access to CCR enabled\n");
  return 0;
```

```
29 }
30
31 void cleanup_module(void) {
32 }
```

B.2 signal.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <stdint.h>
4
5  #include <fcntl.h>
6  #include <sys/mman.h>
7  #include <sys/stat.h>
8  #include <unistd.h>
9
10 #ifndef PI_VERSION
11 #  define PI_VERSION 3
12 #endif
13
14 #define BCM2708_PI1_PERI_BASE  0x20000000
15 #define BCM2709_PI2_PERI_BASE  0x3F000000
16 #define BCM2711_PI4_PERI_BASE  0xFE000000
17
18 // -- General, Pi-specific setup.
19 #if PI_VERSION == 1
20 #  define PERI_BASE BCM2708_PI1_PERI_BASE
21 #elif PI_VERSION == 2 || PI_VERSION == 3
22 #  define PERI_BASE BCM2709_PI2_PERI_BASE
23 #else
24 #  define PERI_BASE BCM2711_PI4_PERI_BASE
25 #endif
26
27 // --- GPIO specific defines
28 #define GPIO_REGISTER_BASE 0x200000
29 #define GPIO_SET_OFFSET 0x1C
30 #define GPIO_CLR_OFFSET 0x28
31 #define PHYSICAL_GPIO_BUS (0x7E000000 + GPIO_REGISTER_BASE)
32
33 #define TIME_LOW   60000
34 #define TIME_HIGH  60000
35
36 //const int GPIO_OUT=22;  // pin 15
37 #define TOGGLE_GPIO 22
38
39 #define PAGE_SIZE 4096
40
41 // Return a pointer to a periphery subsystem register.
42 static void *mmap_bcm_register(off_t register_offset) {
43   const off_t base = PERI_BASE;
44
```

```
45    int mem_fd;
46    if ((mem_fd = open("/dev/mem", O_RDWR|O_SYNC) ) < 0) {
47      perror("can't open /dev/mem: ");
48      fprintf(stderr, "You need to run this as root!\n");
49      return NULL;
50    }
51
52    uint32_t *result =
53      (uint32_t*) mmap(NULL,                  // Any adddress in our space will
          do
54                     PAGE_SIZE,
55                     PROT_READ|PROT_WRITE,    // Enable r/w on GPIO registers.
56                     MAP_SHARED,
57                     mem_fd,                  // File to map
58                     base + register_offset   // Offset to bcm register
59                     );
60    close(mem_fd);
61
62    if (result == MAP_FAILED) {
63      fprintf(stderr, "mmap error %p\n", result);
64      return NULL;
65    }
66    return result;
67  }
68
69  void initialize_gpio_for_output(volatile uint32_t *gpio_registerset, int bit) {
70    // *(gpio_registerset+(bit/10)) &= ~(7<<((bit%10)*3));  // prepare: set as
        input
71    *(gpio_registerset+(bit/10)) |=  (1<<((bit%10)*3));  // set as output.
72  }
73
74  static inline uint32_t ccnt_read (void)
75  {
76    uint32_t cc = 0;
77    __asm__ volatile ("mrc p15, 0, %0, c9, c13, 0" : "=r" (cc));
78    return cc;
79  }
80
81  int main(int argc, char **argv)
82  {
83  uint32_t time_low, time_high;
84
85  time_high = TIME_HIGH;
86  time_low  = TIME_LOW;
87
88  if (argc == 3 )
89  {
90    time_low  = atoi( argv[1] );
91    time_high = atoi( argv[2] );
92  }
93
```

```
94     // Prepare GPIO
95     volatile uint32_t *gpio_port = mmap_bcm_register(GPIO_REGISTER_BASE);
96     initialize_gpio_for_output(gpio_port, TOGGLE_GPIO);
97     volatile uint32_t *set_reg = gpio_port + (GPIO_SET_OFFSET / sizeof(uint32_t))
           ;
98     volatile uint32_t *clr_reg = gpio_port + (GPIO_CLR_OFFSET / sizeof(uint32_t))
           ;
99
100  uint32_t t0;
101  //uint32_t t1;
102
103  while( 1 )
104  {
105    // Enable counter and reset it
106    __asm__ volatile ("MCR p15, 0, %0, c9, c12, 0\t\n" :: "r"(4+1));
107
108    *clr_reg = (1<<TOGGLE_GPIO);
109
110    do {
111      t0 = ccnt_read();
112    } while(time_low > t0);
113
114    // Enable counter and reset it
115    __asm__ volatile ("MCR p15, 0, %0, c9, c12, 0\t\n" :: "r"(4+1));
116
117    *set_reg = (1<<TOGGLE_GPIO);
118
119    do {
120      t0 = ccnt_read();
121    } while(time_high > t0);
122  }
123  }
```

B.3 Makefile

```
1  obj-m += enable_ccr.o
2
3  all:
4    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
5
6  clean:
7    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

Apéndice C

Herramientas y recursos

Las herramientas necesarias para la elaboración del proyecto han sido:

- PC compatible.
- Sistema operativo GNU/Linux [21].
- Raspberry Pi 3 modelo B.
- Lenguaje de programación Shell Scripting.
- Lenguaje de programación C.

Universidad de Alcalá
Escuela Politécnica Superior