Facultad de Ciencias

Comparación de estrategias para mejorar la latencia de planificación en sistemas Linux

(Comparison of strategies to improve scheduling latency in Linux systems)

Trabajo de Fin de Grado
para acceder al

GRADO EN INGENIERÍA INFORMÁTICA

Autor: Iván Revuelta Fernández

Director: Mario Aldea Rivas

Co-Director: Alejandro Pérez Ruiz

Junio - 2019

Contenido

Índice de Figuras

Índice de Tablas

Resumen

Las aplicaciones de tiempo real son imprescindibles en la sociedad actual, esto es debido al gran número aplicaciones que se le está dando a la tecnología en entornos que tienen requisitos temporales, algunos de estos ejemplos son los llamados “coches sin piloto”, implantes auditivos o sistemas de aviación.

En este trabajo, se ha querido estudiar las diferentes alternativas para mejorar el comportamiento temporal de Linux en las aplicaciones de tiempo real y evaluar su rendimiento. Además, debido a lo extendido que está Android en los diversos dispositivos móviles, se ha querido ampliar el estudio a este sistema operativo dado que está basado en el kernel de Linux y las soluciones que se han aplicado son compatibles. Las alternativas para Linux en las que se centra este trabajo son el parche PREEMPT_RT, una popular solución para mejorar el comportamiento temporal de Linux, y una solución propuesta por el grupo de Ingeniería Software y de Tiempo Real de la Universidad de Cantabria, que se basa en un mecanismo disponible en Linux/Android que permite aislar un núcleo del procesador para ejecutar las aplicaciones de tiempo real y disminuir las interferencias con el resto del sistema.

Se ha elegido la Raspberry Pi 3 B como dispositivo para realizar el análisis de las diferentes soluciones, el motivo es que se querían llevar a cabo todas las pruebas en las mismas condiciones y en la Raspberry se puede instalar tanto una distribución de Linux como una de Android. Para estas pruebas se ha estresado el sistema con *benchmarks* como Antutu o Geekbench y se han medido los tiempos de latencia y algunas funciones propias del sistema operativo relacionadas con las aplicaciones de tiempo real. Estás funciones son los bloqueos de *mutex*, cambios de prioridad y mediciones con el *timer*.
Se han realizado unas primeras pruebas donde se han evitado las interferencias con otras aplicaciones de tiempo real, donde el sistema se estresa con los benchmarks nombrados anteriormente. Con este experimento lo que se mide es cómo influye el kernel de Linux y las interrupciones que no se pueden enmascarar en las aplicaciones de tiempo real.
Por otro lado, se han realizado unas pruebas estresando con el sistema con una simulación de una aplicación de mayor prioridad que con la que se realizan las medidas. El objetivo de este segundo experimento es demostrar que el mecanismo de aislamiento es útil cuando hay interferencias con otras aplicaciones de tiempo real.

Palabras clave: Linux, Tiempo Real, Android, PREEMPT_RT, Raspberry PI

Abstract

Real-time applications are essential in today's society, this is due to the large number of applications that are being given to technology in environments that have temporary requirements, some of these examples are the so-called "cars without a pilot", hearing implants or aviation systems.

In this work, we wanted to study the different alternatives to improve the temporal behaviour of Linux in real-time applications and evaluate its performance. In addition, due to Android is extremely used in the different mobile devices, we wanted to extend the study to this operative system since it is based on the Linux kernel and the solutions that have been applied are compatible. The alternatives for Linux that this work focuses on are the PREEMPT_RT patch, a popular solution to improve the temporal behaviour of Linux, and a solution proposed by the Software Engineering and Real Time group of the University of Cantabria, which is based on in a mechanism available in Linux / Android that allows to isolate a nucleus of the processor to execute the applications in real time and to reduce the interferences with the rest of the system.

The Raspberry Pi 3 B has been chosen as a device to perform the analysis of the different solutions, the reason is that they wanted to carry out all the tests in the same conditions and in the Raspberry you can install both a Linux distribution and one of Android For these tests, the system has been stressed with benchmarks such as Antutu or Geekbench, and the latency times and some functions of the operating system related to real-time applications have been measured. These functions are the mutex locks, priority changes and measurements with the timer.
First tests have been carried out where interference with other real-time applications has been avoided, where the system is stressed with the benchmarks mentioned above. With this experiment what is measured is how the Linux kernel influences and the interruptions that can´t be masked in real-time applications.
On the other hand, tests have been performed stressing with the system with a simulation of an application of higher priority than with which the measurements are made. The objective of this second experiment is to demonstrate that the isolation mechanism is useful when there is interference with other real-time applications.

1. Introducción

Las aplicaciones de tiempo son una necesidad en ámbitos como la automoción, la aviación, o incluso en aquellos relacionados con la salud y los implantes médicos[1], es por eso por lo que el objetivo principal de este Trabajo de Fin de Grado es estudiar las diferentes soluciones existentes para acabar con las limitaciones que tiene Linux para dar soporte a las aplicaciones de tiempo real y además analizar y comparar los resultados obtenidos con estas soluciones. En este primer capítulo se hará un acercamiento a los diferentes sistemas operativos y las soluciones elegidas para su estudio.

1.1. Contexto

Desde su creación en 1991, Linux ha sido uno de los sistemas operativos más extendido en el ámbito de la informática. Pero en los últimos años ha evolucionado y se ha adaptado tanto que hasta entre los usuarios que utilizan el ordenador de una forma más básica ha empezado a ser popular este sistema operativo. Aunque Linux se creó como un sistema de propósito general, dada su popularidad, se ha modificado y se ha extendido su uso a otros campos más específicos, como por ejemplo Kali Linux[2] que es un sistema operativo dedicado a la explotación de fallos de seguridad o SteamOS[3] que es una versión de Linux diseñada para la ejecución de videojuegos.

Por otro lado, tenemos Android que es el sistema operativo más utilizado en dispositivos móviles (smartphones, tablets, smartwatch…), se estima que entorno al 80% de ellos utilizan Android como sistema operativo. Android está desarrollado por Google desde que compró la empresa en 2005 y es de código abierto lo que posibilita tener una gran comunidad de desarrolladores que aportan diferentes modificaciones del sistema operativo. Android está basado en el kernel de Linux y por tanto está incluido dentro de los sistemas operativos que vamos a estudiar.
Como podemos observar en la Figura 1, el sistema operativo está compuesto por cinco capas. La primera de las capas es donde están las aplicaciones normalmente desarrolladas en Java, es decir aquellas que permiten interactuar directamente al usuario con el teléfono (Whatsapp, YouTube, Instagram…). En la segunda capa se encuentra el *framework* de Android el cuál es el que permite crear las aplicaciones para Android. La tercera capa es donde están las librerías del core y la máquina virtual Dalvik o ART (Android RunTime) dependiendo de la versión de Android. En la cuarta capa se encuentran las librerías C y C++ que son quienes aportan la mayoría de las características al sistema, entre ellas está *Bionic* de la que hablaremos más adelante, que es una modificación de la librería *glibc*. Y finalmente, como se había adelantado, en su última capa se encuentra el kernel de Linux.

[1] http://implantecoclear.org
[2] https://www.kali.org/
[3] https://store.steampowered.com/steamos/

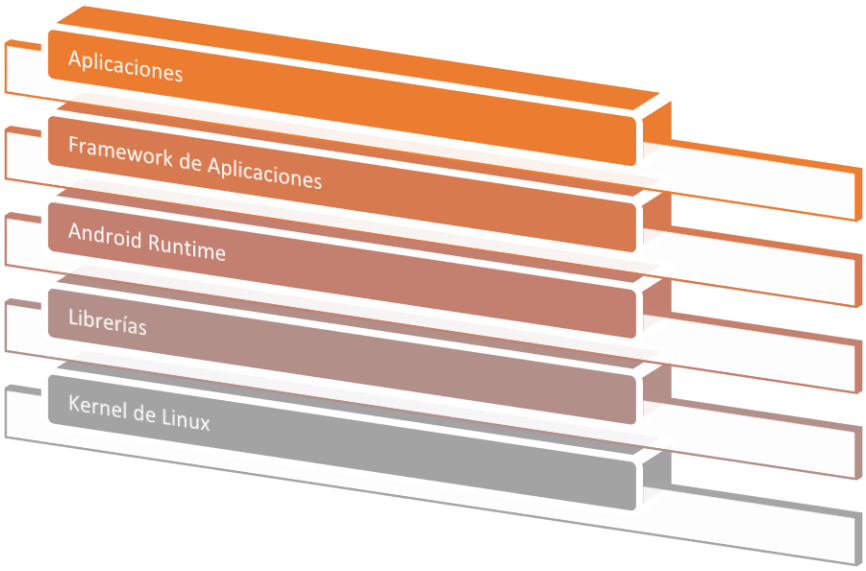

Figura 1. Arquitectura Android

En este trabajo estudiaremos las distintas alternativas que se pueden aplicar sobre el kernel utilizado en Android para otorgarle una mayor predictibilidad temporal.

1.2. Objetivo

Como se ha adelantado en la sección anterior, este Trabajo Final de Grado va a estar centrado en sistemas operativos basados en Linux y más concretamente en diferentes soluciones existentes para mejorar el comportamiento temporal de los sistemas Linux con las aplicaciones de tiempo real. Las soluciones que se van a estudiar son las siguientes:

-La popular solución para añadir a Linux características de tiempo real, el parche PREEMPT_RT [2].

-La activación de ciertas características ya incluidas dentro del kernel de Linux pero que de forma predeterminada están desactivadas. Estás características están relacionadas con el mecanismo Read-Copy Update (RCU) que se explicará en el siguiente capítulo.

-El aislamiento de uno o varios cores de un procesador para ejecutar en ellos las aplicaciones de tiempo real, desarrollado por el grupo ISTR de la Universidad de Cantabria [1]. Este aislamiento conseguirá que las aplicaciones de tiempo real se vean considerablemente menos interferidas por otras aplicaciones o por el propio kernel del sistema operativo.

Por lo tanto, el objetivo principal de este Trabajo Final de Grado es evaluar las distintas alternativas existentes para mejorar el comportamiento temporal de los sistemas Linux y comparar los distintos resultados arrojados. Además, se combinarán las distintas alternativas para comprobar si se producen mejoras sustanciales en los tiempos de

respuesta para las aplicaciones de tiempo real. Por otro lado, como objetivo secundario tenemos la realización de la misma evaluación, pero para el sistema operativo Android.

2. Antecedentes

En este capítulo se expondrán algunas cuestiones relevantes del kernel de Linux, como el planificador de tareas, el cuál es el encargado de repartir el tiempo de computo entre las distintas tareas que necesitan ejecutarse en el sistema operativo. Además, se explicarán detalladamente las soluciones que se indicaron en los objetivos: el parche PREEMPT_RT, el aislamiento [1] y el mecanismo RCU. Finalmente se explica cómo se han incorporado estas soluciones a Android.

2.1. Planificador de Linux

Linux es un sistema operativo multitarea, por tanto, puede ejecutar varios threads concurrentes a la vez. Para ello el kernel debe garantizar que estos threads tengan un tiempo de CPU, quién se encarga de esto es el planificador de tareas. Con el objetivo de organizar los threads según la importancia de estos, se les asigna una prioridad la cual puede ir desde 0 hasta 139. Según la prioridad que se le asigne a un thread, estará en el grupo de los threads de tiempo real, para los que se reservan las prioridades 0-99, o en el grupo de los threads de usuario, que son las que su prioridad está comprendida entre 100-139.

Los threads del segundo grupo son administrados por el "Planificador Completamente Justo" (Completely Fair Scheduler en inglés) que fue creado por Ingo Molnar y sustituyó al planificador O(1) en la versión del kernel de Linux 2.6.23. El Planificador Completamente Justo en vez de asignar una fracción de tiempo a cada uno de los threads, lo que hace es calcular el tiempo que le corresponde en función del número de procesos preparados para ser ejecutados. El planificador consigue la "justica" dejando que un thread se ejecute un tiempo proporcional dividido entre el peso total de todos los procesos preparados para ser ejecutados, obviamente un thread con mayor prioridad tendrá más peso.

Por el otro lado tenemos el grupo de los threads de tiempo real, cuando uno de estos threads esté preparado para ejecutarse obtendrá la CPU siempre y cuando no haya otro thread de tiempo real con una prioridad mayor. Los threads de tiempo real pueden tener tres políticas:

-SCHED_FIFO: En este caso el thread que primero llegue, primero se ejecutará hasta que haya finalizado, hasta que llegue otro con prioridad mayor o hasta que libere la CPU voluntariamente.

-SCHED_RR: Es igual que SCHED_FIFO, pero en esta opción se utilizan las fracciones de tiempo, es decir, cuando dos threads tengan la misma prioridad se repartirán la CPU usando *round robin*. Por tanto, un thread se ejecutará hasta que agote su quantum o hasta que un thread de mayor prioridad lo expulse del procesador.

- SCHED_EDF: Esta tercera política es la más reciente y se ha incluido en la versión 3.14 del kernel Linux, SCHED_EDF está basada en el algoritmo *Earliest Deadline First* [3]. Esta política otorga el procesador con el plazo de finalización más cercano y, además,

una tarea que tenga esta política asignada tendrá una prioridad mayor que otra con SCHED_FIFO o SCHED_RR asignados.

Hay que aclarar que no es cierto que una tarea de tiempo real puede disponer del procesador hasta que finalice, aunque no haya ninguna otra tarea con mayor o igual prioridad. Esto sí que pasaría en un sistema de tiempo real puro, pero Linux tiene un mecanismo para evitar que una tarea de tiempo real consiga bloquear el sistema. Este mecanismo está condicionado por dos variables: *rt_period* y *rt_runtime*. *rt_period* indica el tiempo que dura un periodo y *rt_runtime* el tiempo que pueden utilizar las tareas de tiempo real en un periodo. Por defecto, *rt_period* es igual a 1s y *rt_runtime* es igual a 0,95s, por lo que una tarea de tiempo real como máximo podría disponer del 95% del procesador.

Para las tareas de tiempo real es importante que puedan conseguir tiempo de CPU en cuanto están listas para ejecutarse, como se verá en la siguiente sección, el parche PREEMPT_RT ayuda a que esto ocurra convirtiendo casi todo el kernel en secciones de código expulsables.

2.2. Parche PREEMPT_RT

El parche PREEMPT_RT [2] es la forma más popular de mejorar las características de los sistemas Linux para se adapten a las aplicaciones de tiempo real. El parche es actualizado y mantenido por un grupo de desarrolladores dirigidos por Ingo Molnar de quién ya hemos hablado dado que también es el creador del Planificador Completamente Justo. De una forma muy general lo que se consigue aplicando el parche es que la mayoría del kernel sea expulsable a excepción de las regiones críticas protegidas por *raw_spinlock*. Esto permite que cuando una tarea de tiempo real con una prioridad alta pueda comenzar a ejecutarse sin tener que esperar a no ser que haya otra de mayor prioridad ejecutándose.

En primer lugar, el parche cambia la forma en la que están implementados los mecanismos de bloqueo de dentro del kernel y usando *rtmutexes*. Con esto consigue que los mecanismos de bloqueo en el kernel sean totalmente expulsables. Para conseguir que una zona crítica sea de verdad no expulsable deberíamos usar *raw_spinlock*.

El parche también soluciona el problema de la inversión de prioridades [4], este problema consiste en que, por necesidad de acceso a una sección crítica, la cual ya está bloqueada por una tarea de prioridad baja, una tarea de alta prioridad acaba siendo de baja prioridad.

El parche se encarga de convertir a los controladores de interrupciones en hilos del kernel, lo cual les hace ser expulsables, y trata a las interrupciones suaves en el contexto del kernel el cual se representa con una *task_struct* como si fuera un proceso de usuario. Por último, otro de los cambios relevantes que hace el parche es separar la API del *timer* de Linux en dos, una para los *timers* de alta precisión y otro para los tiempos de espera. Así, el *timer* de alta precisión consigue una resolución del orden de nanosegundos

Para conseguir esta solución en Linux, simplemente hay que descargar el código fuente del kernel que necesitamos y comprobar que sea compatible con el parche. Además, si se quiere ahorrar el esfuerzo de aplicar el parche se pueden descargar las fuentes del kernel

con el parche ya aplicado del GitHub oficial de Linux. Por último, solo habría que compilar el kernel activando la opción CONFIG_PREEMPT_RT_FULL.

2.3. RCU

En la sección de objetivos se habló del mecanismo Read-Copy-Update (RCU), el cual mediante la activación de algunas opciones del kernel puede ayudar a mejorar las latencias. En primer lugar, se explicará en qué consiste el mecanismo RCU y, en segundo lugar, se indicarán los parámetros modificados en el kernel.

RCU (Read-Copy Update) es un mecanismo de sincronización que permite a los lectores acceder a recursos críticos a la vez que están siendo actualizados, de esta forma se consigue que las lecturas sean mucho más rápidas. Esto es así porque es el escritor quién debe verificar que no hay ningún lector operando sobre el dato que va a actualizar.

Los parámetros modificados son estos:

- CONFIG_RCU_BOOST: Cuando esta opción está activada se aumenta la prioridad de los lectores que están bloqueados en mitad de una lectura.

- CONFIG_RCU_BOOST_PRIO: Otorga la prioridad de tiempo real indicada a los lectores que han sido expulsados.

- CONFIG_RCU_BOOST_DELAY: Especifica durante cuánto tiempo el mecanismo RCU permitirá que un período de gracia (para el escritor) se retrase antes del aumento de la prioridad de los lectores bloqueantes.

- CONFIG_TREE_PREEMPT_RCU: Activa la implementación RCU basada en árbol que es apropiada para sistemas de multiprocesamiento simétrico de tiempo real y con baja latencia. También es recomendada para su para sistemas con numerosos procesadores, aunque también escala razonablemente bien en sistemas más pequeños.

Para activar estos parámetros hay que modificar el fichero *.config*, después compilar el kernel e instalar dicho kernel en el sistema operativo. Una vez indicados los parámetros que se han activado para realizar las pruebas, se va a explicar en qué consiste este mecanismo.

2.4. Aislamiento

El aislamiento de los núcleos del procesador es otra de las soluciones comentadas en la Sección 1.2 del capítulo anterior. Con esta solución lo que se propone es que las aplicaciones de tiempo real se ejecuten en un núcleo del procesador reservado para ellas y en el conjunto restante de núcleos se ejecuten las aplicaciones de usuario y del sistema. El grupo ISTR de la Universidad de Cantabria analizó dos formas que incluye Linux de manera natural para aislar los núcleos del procesador [1]. Estas dos soluciones son

isolcpus y *cpusets*. Concluyeron que *cpusets* era una mejor forma de aislar los núcleos del procesador por dos motivos, el primero es que para aplicar *isolcpus* solo se puede durante el sistema de arranque mientras que *cpusets* se puede aplicar en cualquier momento una vez se esté ejecutando el sistema. Y, en segundo lugar, cualquier tarea que trate de modificar su afinidad usando *sched_affinity* o el comando *taskset* puede añadir un proceso al núcleo aislado si estamos usando *isolcpus* pero en el caso de *cpusets* solo se podrá añadir un proceso escribiendo su PID en un fichero que determina que procesos pueden usar el núcleo aislado.

Además de los procesos, también hay que aislar el núcleo de las interrupciones que puedan afectar a nuestras aplicaciones de tiempo real. Para ello, hay que activar una máscara que se encuentra en el archivo *smp_affinity* con la que desviaremos todas las interrupciones al núcleo seleccionado, que será uno de los que no queremos aislar. Pero debemos tener en cuenta que no todas las interrupciones se pueden enmascarar.

En este articulo también se describen una serie de puntos que hay que tener en cuenta si queremos aplicar este mecanismo:

- Hay que fijar la frecuencia del procesador para evitar los cambios de frecuencia y conseguir la predictibilidad necesaria en los tiempos de respuesta. Así el sistema será más estable lo cual es óptimo para las aplicaciones de tiempo real.

- Hay que activar las opciones del kernel CONFIG_CPUSETS y CONFIG_PREEMPT. La primera nos permitirá crear nuestros propios *cpusets* y controlarlos, y la segunda hace que el código del kernel sea mayoritariamente expulsable.

- En el caso de Android, hay ciertos demonios que apagan los núcleos del procesador para el ahorro de energía, por lo que hay que desactivarlos.

Finalmente, en el artículo se hace una comparación entre los resultados de un dispositivo con Android básico y otro dispositivo con Android aislado, en dichos resultados se comprueba que el Android aislado obtiene una respuesta temporal significativamente mejor que un Android donde no se aplican los mecanismos descritos anteriormente. En este Trabajo Final de Grado se repetirán estas pruebas con el objetivo de verificar los resultados, compararlos y combinarlos con las otras soluciones. Aunque esta solución es útil tanto para Android como para Linux, en el caso del primero se sigue teniendo unas limitaciones más estrictas causadas por la necesidad de ahorrar espacio de memoria y batería debido a las características de los dispositivos donde se ejecuta. Estas limitaciones son mitigadas añadiendo a esta solución algunos cambios extra que se comentarán en la siguiente sección.

2.5. Limitaciones de la librería Bionic

Centrándonos en las soluciones Android, en un estudio previo [5] también realizado por el grupo ISTR de la Universidad de Cantabria donde hacían un resumen de las diferentes soluciones para convertir Android en un sistema para aplicaciones en tiempo real. Debido a la complejidad de adaptar y actualizar las soluciones, el grupo ISTR intentó dar una

solución más sencilla para conseguir un buen rendimiento de las aplicaciones de tiempo real en Android.
La solución propuesta en este caso se combina con lo que se explicó en la Sección 2.4, que mediante esta técnica conseguíamos que no hubiera interferencias con el resto del sistema cuando una aplicación de tiempo real se estaba ejecutando, se enmascaraban todas las interrupciones posibles para que se desviaran a otro procesador, se fijaba la frecuencia de los procesadores y se desactivaban sus posibles mecanismos de apagado. El añadido de este estudio es que resuelve las limitaciones que tiene la librería *Bionic*, ya que esta es una librería adaptada a los dispositivos móviles donde el almacenamiento es limitado y los procesadores no son tan veloces como un sobremesa o portátil. Por lo que en la adaptación se eliminaron algunas funciones relacionadas con las aplicaciones de tiempo real como por ejemplo las relacionadas con la asignación de prioridades tanto a *mutex* como a threads. Para resolver esto, el grupo ISTR carga la librería *glibc*, que es la librería estándar que se utiliza en Linux, en el sistema Android para hacer uso de ella. Y para que las aplicaciones compiladas puedan utilizar esta librería se indica en la orden de compilación cuál es la ruta de las librerías dinámicas en el dispositivo de Android y donde se encuentra el enlazador dinámico.

```
arm-linux-gnueabi-gcc hello_world.c -o hello_world
-Wl,--dynamic-linker=/data/local/libs/ld-linux.so.3
-Wl,-rpath=/data/local/libs -fPIE -pie
```

Figura 2. Compilación dinámica indicando la localización de la libreria glibc [5].

Dando por completa la explicación de estos mecanismos, se pasará a explicar en las siguientes secciones las soluciones que se indicaron al principio de esta sección que, aunque son más compleja han sido tenidas en cuenta.

2.6. RTAndroid

RTAndroid [6][7] es la primera de las soluciones comentadas en la sección anterior que ha sido desarrollada en la Universidad de Aachen, Alemania. Esta solución utiliza el parche PREEMPT_RT para otorgar una mayor predictibilidad temporal al kernel de Linux. La aplicación de este parche en Android no se puede realizar de forma directa, ya que el kernel de este sistema no sigue la línea de desarrollo principal usada por Linux. Por lo tanto, se debe de adaptar el parche para que pueda ser aplicado sobre el kernel de Android.

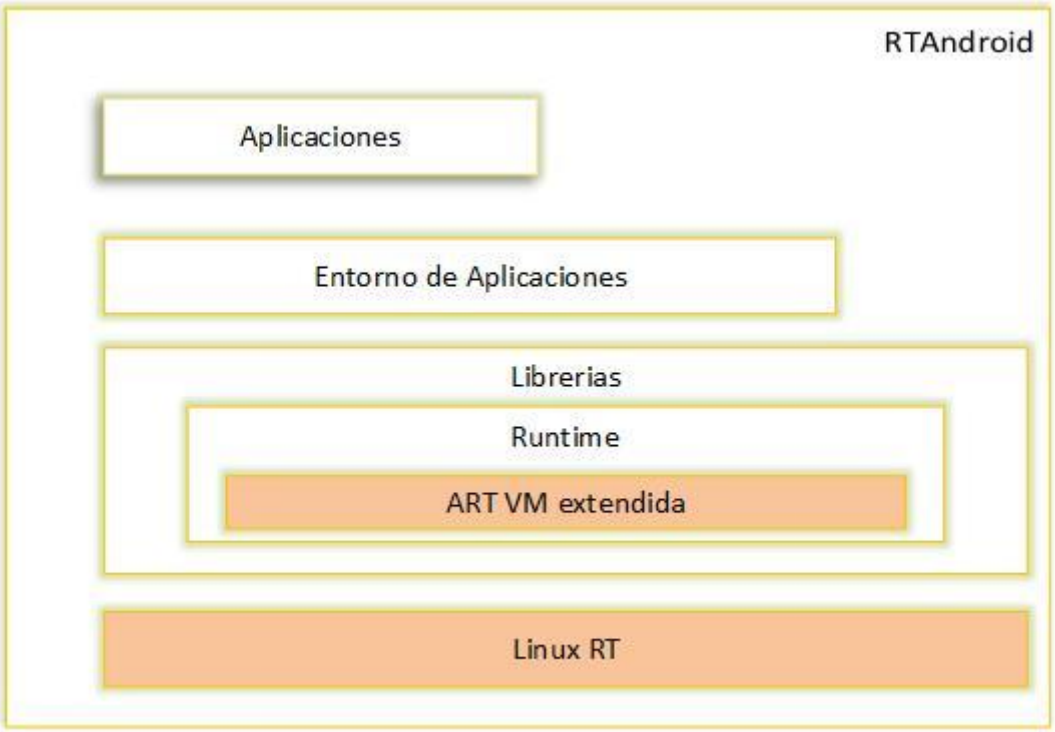

Figura 3. Arquitectura RTAndroid [6], las partes coloreadas son los cambios respecto a la arquitectura Android.

Además de la adaptación del parche PREEMPT_RT, como se puede observar en la Figura 3 también han modificado la máquina virtual de Dalvik y el recolector de basura. Lo que se consigue con estos cambios es que hacen que todo lo relacionado con la máquina virtual sea más predecible. Aunque esto último no va a afectar a la evaluación llevada a cabo en este trabajo ya que no se han realizado pruebas sobre la máquina virtual Java ni sobre el entorno de aplicaciones. Por último, su principal desventaja como se ha adelantado es que hay que hacer un gran esfuerzo de actualización con cada nueva versión de Android, aunque este esfuerzo no es tan grande como el que hay que realizar en la siguiente solución.

2.7. RTDroid

Otra de las soluciones propuestas para el estudio es RTDroid [8], una solución implementada el departamento de Ciencias de la Computación e Ingeniería de la Universidad de Buffalo. En este caso, decidieron crear un sistema operativo desde cero con una arquitectura un poco diferente a la original de Android. En la capa del *framework* de aplicaciones han rediseñado tres componentes para que soporten mejor las aplicaciones de tiempo real que son Looper-Handler, Alarm-Manager y RT Sensor Manager. Por otro lado, han añadido una máquina virtual de tiempo real y modificado la librería *Bionic* añadiéndole características de tiempo real. Y, por último, han sustituido el kernel por uno de tiempo real.

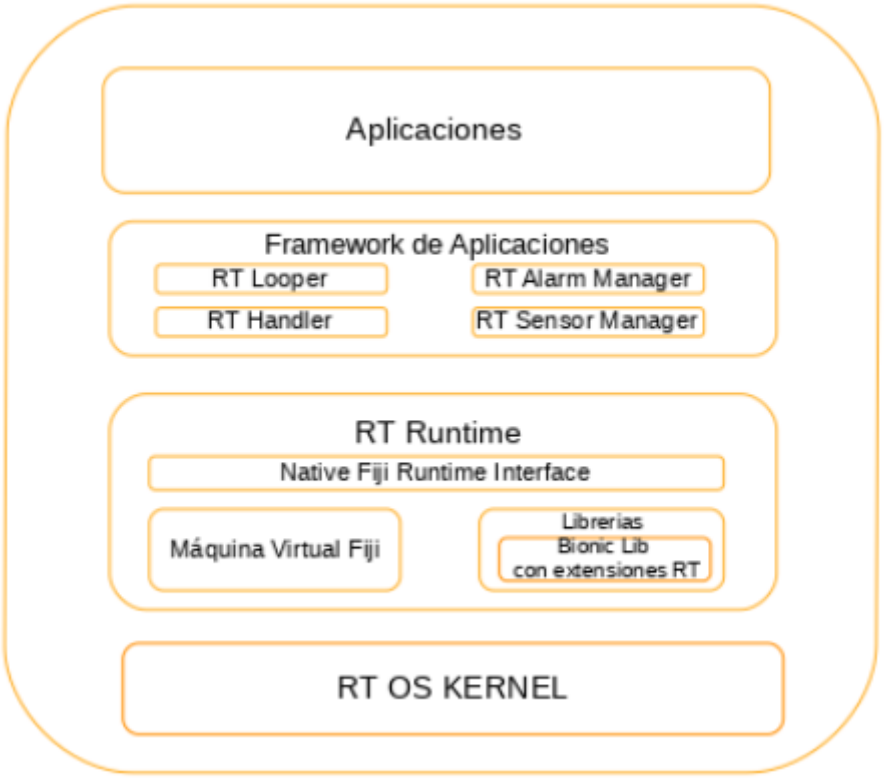

Figura 4. Arquitectura RTDroid [8].

Desgraciadamente no se ha podido acceder ni al código fuente, ni a una imagen de este sistema operativo por lo que no se han realizado las pruebas.

2.8. Android en Raspberry Pi 3 B

Esta última sección de este capítulo se hablará de otro Trabajo de Final de Grado [9], en el cual se hacía una evaluación de la combinación de soluciones [1][5] propuestas por el grupo ISTR de la Universidad de Cantabria comentados anteriormente, en este trabajo se mide la latencia del sistema operativo Android estresado con diferentes *benchmarks* y a modo de representación final se creaba un demostrador en el que mediante un programa se controlaban unos motores de Lego que representaban un giroestabilizador de un solo eje. Para poder utilizar las piezas de Lego, se cargaban las librerías I2C en el sistema Android, las cuales se utilizan como controlador del bus. Se ha tenido en cuenta este trabajo porque en él se recopila como instalar Android en una Raspberry Pi 3 B que cómo se verá más adelante es la máquina donde se realizarán las pruebas.

3. Heramientas.

Antes de comenzar las pruebas, durante este capítulo se detallarán las herramientas utilizadas durante el presente trabajo, y se explicará detalladamente el entorno en el que se han realizado las pruebas.

3.1. Cyclictest

Cyclictest es un programa muy reconocido para la medida de latencias en sistemas operativos de tiempo real, a pesar de que su código tiene un número de líneas que se acerca a las 2000, el algoritmo que utiliza para la medida es bastante simple.

```
1. while(!shutdown){
2.          clock_gettime(par->clock, &now);
3.
4.          clock_nanosleep((par->interval));//default 1000 us
5.
6.          clock_gettime((&next));
7.
8.          calc_diff(&now, &next);
9.
10.               //Update stats
11.
12.       }
```

Cyclictest permite elegir una gran variedad de parámetros, pero para realizar los test solo se han utilizado unos pocos. A continuación, se indicará el comando usado para las pruebas y se explicarán las opciones elegidas.

```
sudo cyclictest -l200000  -p90  -h20000   -t  -a  -m  -n  -q
```

-l[NUM], permite elegir el número de ciclos que se ejecutará el programa indicado por NUM.

-h[NUM], permite crear un histograma de la ejecución donde NUM es la latencia máxima que se guardará en el histograma.

-p[NUM], permite elegir la prioridad con la que se ejecutará el programa, siendo NUM un número entre 0-99.

-m, es una opción recomendada por los creadores que hace que el programa reserve memoria para no ser paginado fuera.

-t[NUM], con este parámetro podemos elegir el número de threads que se ejecutarán, si dejamos la variable NUM en blanco, se ejecutará un thread por CPU.

-a, hará que los threads se ejecuten en el CPU que nosotros queramos, si se combina con -t, se consigue ejecutar un thread en cada uno de los CPUs.

-n, con esto escogeremos que se utilice clock_nanosleep en vez del timer del POSIX.

-q, hará que no se muestre nada hasta el final del programa, donde se mostrará un pequeño resumen.

3.2. Compilador Cruzado

El compilador cruzado es una herramienta fundamental cuando se trabaja con dispositivos como móviles o la Raspberry porque compilar programas en ellos es complicado. Entonces la solución más simple es compilar desde un ordenador convencional (mucho más potente) en el cuál es más sencillo realizar esta tarea. Pero nos encontramos con el problema de que no podemos compilar los programas como si lo hiciéramos para ese mismo ordenador dado que estos dispositivos tienen una arquitectura diferente y aquí es donde aparece el compilador cruzado. Se ha utilizado el *gcc-arm-linux-gnueabihf* de Linux el cual nos permite compilar programas desde un ordenador a un dispositivo con arquitectura ARM.

3.3. Entorno de Pruebas

En esta sección se describirá el entorno de las pruebas, es decir, el dispositivo elegido para estas, las versiones del sistema operativo elegido para realizarlas, los programas que se han elegido para estresar el sistema y el programa con el que se han tomado las medidas.

Como se comentó en el Capítulo 2, se han querido realizar todas las pruebas sobre la misma arquitectura y máquina, ya fueran para Linux o Android. Por eso el dispositivo elegido para llevar a cabo las pruebas es la Raspberry Pi 3 B. La Raspberry Pi 3 B es un miniordenador diseñado para fomentar la educación en programación, por eso su tan reducido precio que está entorno a los 35€. Las características principales en este dispositivo son bastante simples, cuenta con un procesador de 4 nucleos de arquitectura ARM con una frecuencia de 1,2Ghz y una memoria RAM de 1 GB.

Los sistemas operativos en los que se han realizado las pruebas son Raspbian y Android. Raspbian es una distribución de Linux diseñada para la Raspberry Pi 3, y en concreto la versión del kernel sobre la que se han realizado las pruebas es la 4.14. Mientras que para Android se ha escogido la versión *Nougat* que corresponde con la versión 7.1 y está desarrollada sobre la versión 4.4.1. del kernel de Linux.

En cuanto a como se ha cargado el sistema, se han utilizado diferentes opciones dependiendo si el sistema operativo era Raspbian o Android. En el caso de Raspbian, se

ejecutaba el *benchmark* de Geekbench[4] y a la vez un video de YouTube con la calidad a 1080p. Mientras que para Android que solo permite una aplicación en primer plano, se ejecutaba el *benchmark* de Antutu[5], que es el más popular para los dispositivos móviles y en base a el cuál se establece un completo ranking con la puntuación de cada uno de los dispositivos.

Finalmente, se va a detallar cuales son los programas con los que se han recogido las medidas. Aunque en este mismo capítulo se ha hablado de Cyclictest que es un programa para la medida de latencias y orientado a sistemas con características de tiempo real, solo se ha utilizado de manera complementaria. El programa principal que se ha usado para medir la latencia es el mismo que se utilizó en el Trabajo de Fin de Grado [9] previo del cuál se habló en el capítulo anterior. A partir de este momento se utilizará "Programa de medida de latencia" para referirse a este programa de una forma más cómoda. El algoritmo del programa de medida de latencia es similar al algoritmo del Cyclictest pero con una leve diferencia, en este caso no se utiliza la función *clock_nanosleep()* como se observa en las siguientes líneas de código.

```
1. while(i<size){
2.          clock_gettime(CLOCK_MONOTONIC, &time);
3.          n2 = time.tv_nsec + time.tv_sec*NANOS;
4.          medida = n2 - n1;
5.          n1 = n2;
6. }
```

Lo que se consigue con este programa, es que si otra tarea nos expulsa del procesador quedará reflejado ya que la diferencia en tre los tiempos *n1* y *n2* será más grande. Como lo que se quiere medir es la latencia para aplicaciones de tiempo real, el programa se lanza con la política *SCHED_FIFO* y con la máxima prioridad para esta política que es 99.

Por otro lado, también se han querido medir ciertos aspectos de caracterización de los sistemas operativos, como algunas funciones de los *mutex*, *timer* y cambios de prioridad. Para recoger estás medidas también se ha utilizado un algoritmo similar al del programa de medida de latencia.

```
1. while(i<size){
2.          clock_gettime(CLOCK_MONOTONIC, &time);
3.          n1 = time.tv_nsec + time.tv_sec*NANOS;
4.          //Aqui la function que queremos medir
5.          pthread_setschedprio(t, j);
6.          n2 = time.tv_nsec + time.tv_sec*NANOS;
7.          medida=n2-n1;
8. }
```

[4] https://www.geekbench.com/
[5] http://www.antutu.com/en/ranking/rank1.htm

Por último, se listarán las funciones de las que se han recogido medidas:

- *Mutex:*
  - *lock()*
  - *trylock()*
  - *unlock()*
- *Timer*
  - *clock_nanosleep()* – Relativo
  - *clock_nanosleep()* – Absoluto
- Cambios de Prioridad
  - *pthread_setschedparam()*
  - *pthread_setschedprio()*

4. Evaluación de las soluciones

En este capítulo se tratará de evaluar las soluciones bajo estudio, es decir las que se explicaron en el Capítulo 2, y las combinaciones que se han hecho entre ellas. Para cada solución se mostrarán sus resultados de latencia y sus resultados de caracterización. En las secciones 4.1 y 4.2 los resultados de la latencia han sido obtenidos mediante el Programa de medida de latencia el cuál será lanzado con la prioridad máxima para así evitar las interferencias con otras aplicaciones de tiempo real. Por lo que los resultados de estas dos secciones solo estarán influenciados por el kernel y las interrupciones que no han podido ser enmascaradas. En la Sección 4.3, se realizará un experimento para comprobar cómo se comporta una aplicación de tiempo real en cada una de las soluciones cuando hay interferencias con otras aplicaciones de tiempo real.

En primer lugar, en esta introducción se intentará aclarar todas las combinaciones estudiadas, para eso se separarán en función del sistema operativo.

Soluciones Raspbian

En este caso, las soluciones aplicadas son el aislamiento, el mecanismo RCU y el parche PREEMPT_RT. Cuando Raspbian tenga el parche aplicado pasará a llamarse RaspbianRT para que sea más fácil referirse a él. Se van a presentar dos árboles con todas las soluciones que se han aplicado en Raspbian, uno de los árboles será para Raspbian simple y el otro para RaspbianRT. Se ha decidido separar así las soluciones porque el parche PREEMPT_RT es la solución que más cambios introduce en el kernel del sistema operativo.

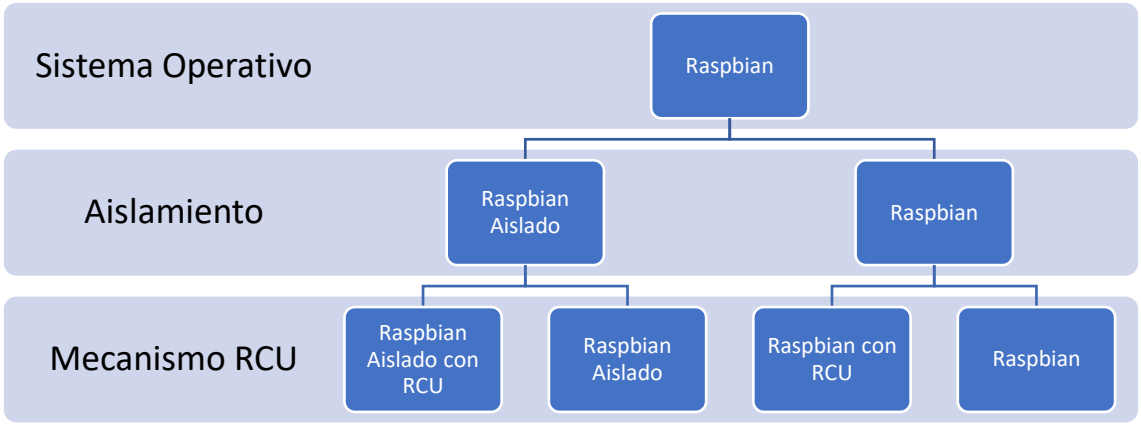

Figura 5. Soluciones Raspbian

En la Figura 5 es un árbol en el que se han representado todas las soluciones para Raspbian, en cada uno de los niveles está propuesta una solución, que se aplicará al hijo de la izquierda de cada nodo en ese nivel y no se aplicará para el hijo de la derecha. En el último nivel se tienen las soluciones aplicadas en Raspbian y como nos referiremos a ellas. Y de forma similar tenemos en la Figura 6 el árbol para las soluciones de RaspbianRT, el cuál recordemos que ya tiene el parche aplicado. Sumando las soluciones de los árboles se tiene que en total se probarán 8 combinaciones para Raspbian, que son: Raspbian, Raspbian aislado, Raspbian con RCU, Raspbian aislado con RCU, RaspbianRT, RaspbianRT aislado, RaspbianRT con RCU y RaspbianRT aislado con RCU

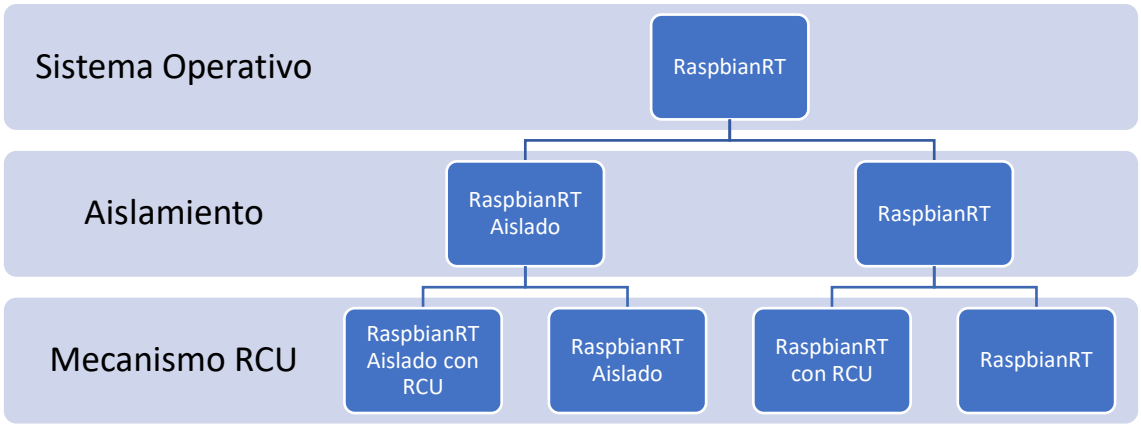

Figura 6. Soluciones RaspbianRT

Soluciones Android

Por otro lado, en el caso de estudio de Android solo se han estudiado dos de las tres soluciones posibles, esto ha sido porque no se disponía de la solución RTAndroid sin compilar para poder añadir las opciones RCU, por lo que esta última opción se ha descartado. Por lo que las soluciones que se van a probar son: Android, Android Aislado, RTAndroid y RTAndroid Aislado.

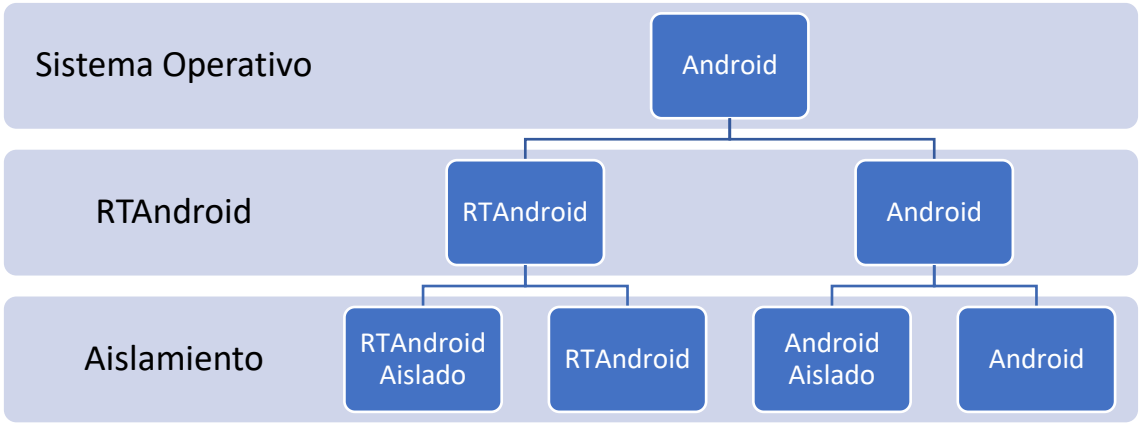

Figura 7. Soluciones Android

4.1. Evaluación de las soluciones Linux

Con el objetivo de establecer una base de lo que es capaz de conseguir Linux sin añadir ninguna de las soluciones anteriores, primero vamos a analizar los resultados obtenidos en Raspbian. Recordemos que las medidas serán con el Programa de medida de latencia al no ser que se indique otra cosa y para la ejecución de estas pruebas el sistema se encuentra cargado con la visualización de un video a 1080p y el *benchmark* de Geekbench. A continuación, se irá probando el resto de las combinaciones con Raspbian y después se pasará a las soluciones que incluyen RaspbianRT.

Evaluación Raspbian

En la siguiente gráfica de la Figura 8 se muestran las latencias obtenidas en 10^9 ciclos. En el eje X se presentan las franjas temporales (en nanosegundos) en las que pueden estar los resultados de (0-1000], (1000-5000], (5000-8000] etc… Y en el eje Y el número de medidas en cada una de las franjas.

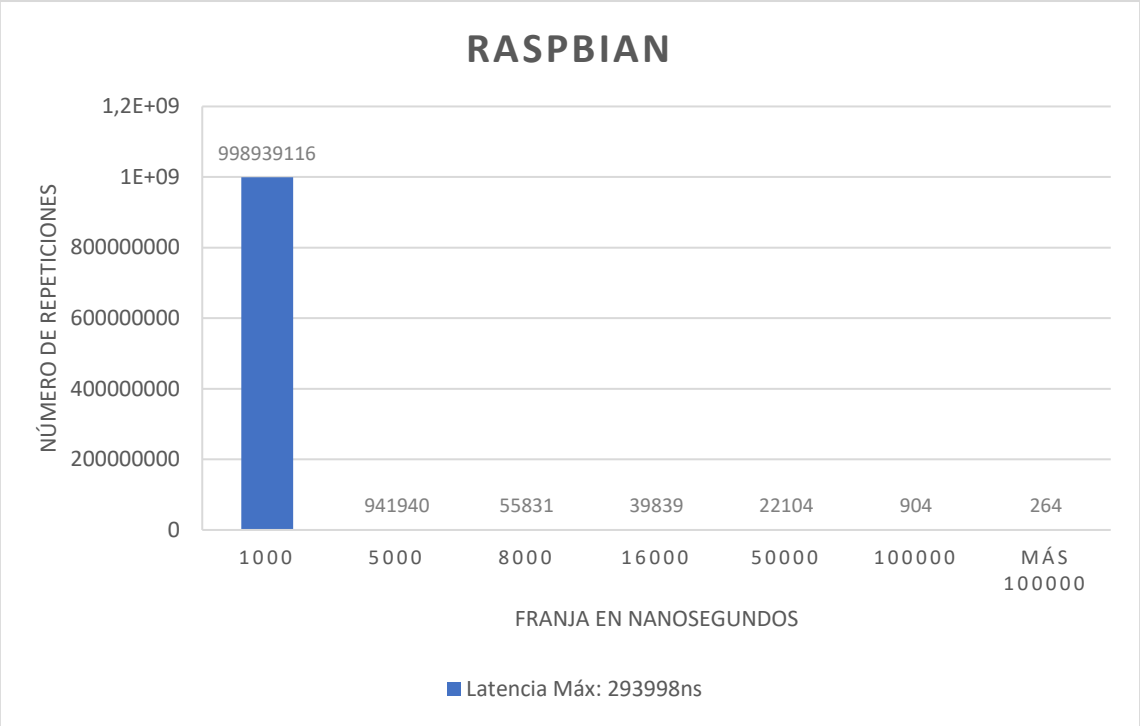

Figura 8. Gráfico de las latencias de Raspbian.

Como se observa en la gráfica, el 99.89% de los resultados son iguales o menores 1000 nanosegundos. Y la latencia máxima es 293998 nanosegundos, es un dato importante dado que en las aplicaciones en tiempo real es importante conocer cuál es el peor de los casos y mejorarlo, incluso por encima de mejorar la latencia media.

Evaluación de Raspbian Aislado frente a Raspbian

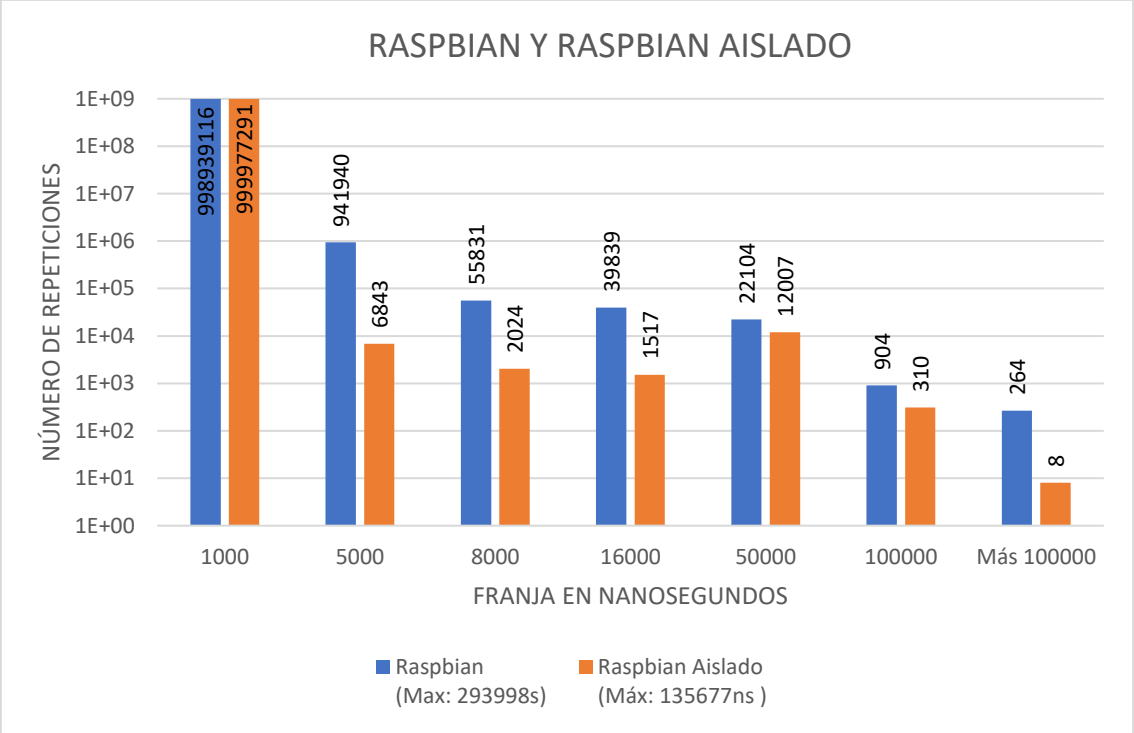

Figura 9. Comparación de Raspbian y Raspbian Aislado en escala logarítmica.

A continuación, en la gráfica de la Figura 9, podemos ver una comparación de los tiempos conseguidos en Raspbian frente a los conseguidos en Raspbian aislado. En las gráficas de comparaciones los resultados se muestran en escala logarítmica. En este caso, Raspbian aislado acumula el 99,99% de los resultados en la primera franja (1000ns) mientras que Raspbian solo acumula el 99,89%, y aunque puede parecer una diferencia muy pequeña, dado al gran número de ciclos con los que se está ejecutando el Programa de medidas de latencia ese 0,1% representa a un millón de resultados que han sido desplazados a las latencias más altas. Por lo que como se puede ver en las latencias más altas, hay una mayor cantidad de resultados de Raspbian que de Raspbian Aislado, donde por ejemplo en la última franja el resultado es 8 mientras que en Raspbian es de 264. Además, esto también se ve reflejado en las latencias máximas, donde hay una diferencia de más de 150000ns, ya que la de Raspbian es 293998ns y la de Raspbian aislado es de 135677ns. Así que en esta comparación se concluye que Raspbian Aislado da un mejor soporte a las aplicaciones de tiempo real que Raspbian.

Evaluación de Raspbian con RCU frente a Raspbian.

El siguiente paso que se debe dar, es comprobar si la solución de los parámetros RCU del Kernel afecta positiva o negativamente a las aplicaciones de tiempo real. Para ellos vamos a realizar la misma comparación anterior, pero en esta ocasión con la solución RCU.

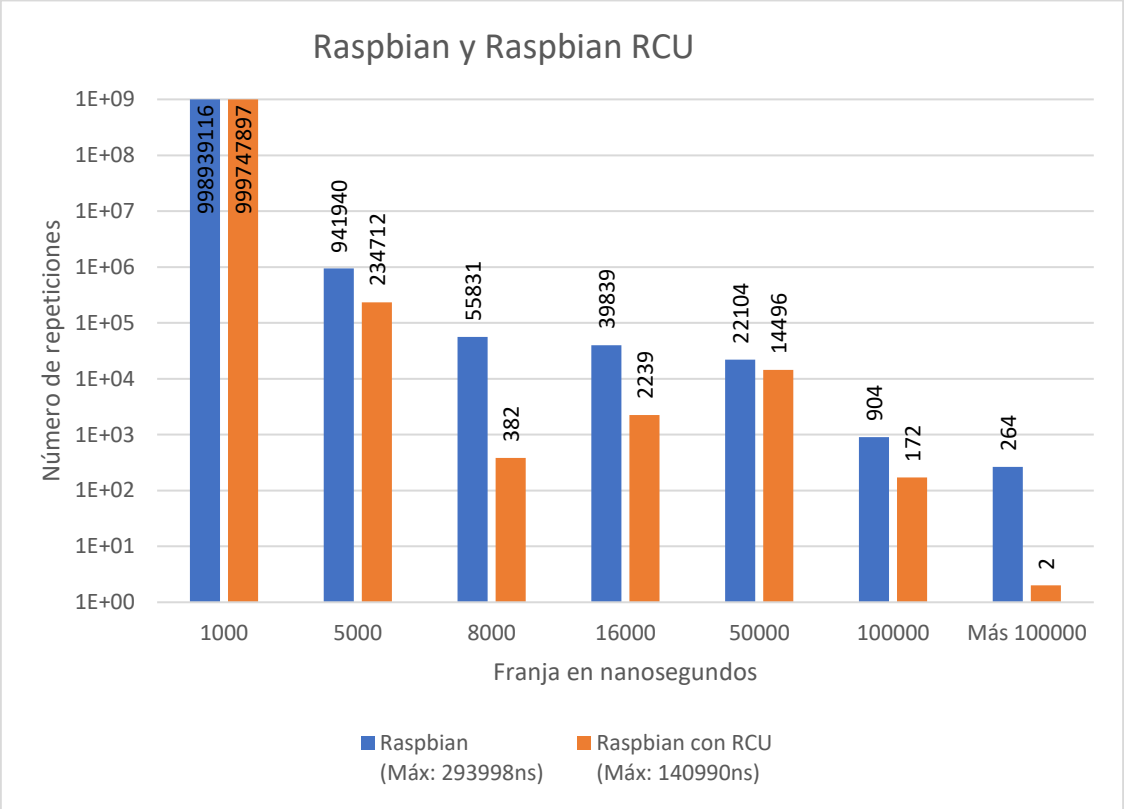

Figura 10. Comparación de Raspbian y Raspbian con RCU en escala logarítmica.

Como podemos ver en la gráfica de la Figura 10, los resultados en la franja más baja parecen no tener una diferencia visual, pero si nos fijamos en los porcentajes Raspbian

acumula un 99,89% en la primera franja y Raspbian con RCU acumula 99,97%, que como se explicó antes esto provoca que un gran número de medidas tengan latencias más bajas en el caso de RCU. Y además en el caso de la latencia máxima se sigue produciendo una gran mejora de más de 150000ns como en el caso anterior siendo la latencia máxima de Raspbian con RCU 140990ns. Así que el mecanismo RCU también parece mejorar los tiempos para las aplicaciones en tiempo real.

Resumen de latencias para las soluciones Raspbian

Cómo último caso vamos a realizar una comparación de todas las soluciones para Raspbian sin aplicar el parche PREEMPT_RT, en dicha comparación vamos a incluir las soluciones que hemos visto hasta ahora y la combinación de ambas.

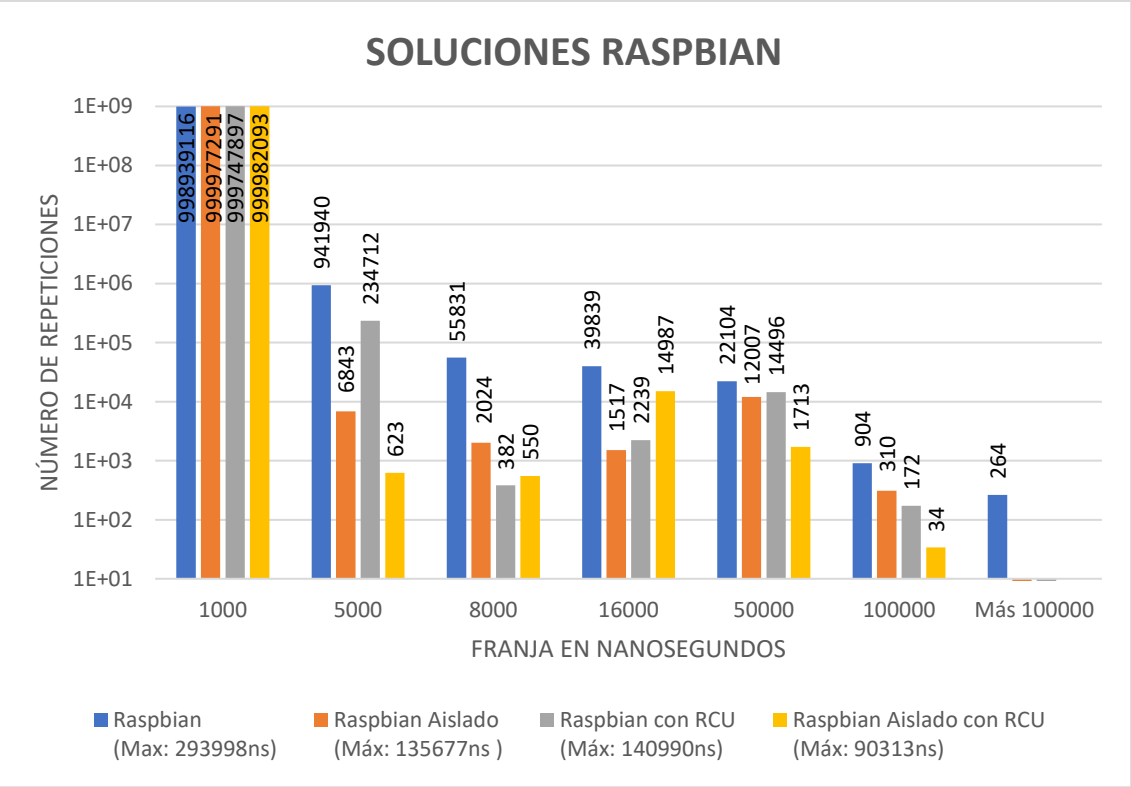

Figura 11. Comparación de las soluciones para Raspbian en escala logarítmica.

En este caso, se puede ver como hay una clara vencedora entre estas soluciones, la combinación de aislamiento y RCU que consigue acumular el 99,99% de los resultados en la primera franja, y es la primera de estas soluciones que consigue no tener ninguna latencia por encima de los 100000ns, siendo 90313ns su latencia máxima. Y Entre la solución de Raspbian Aislado y Raspbian RCU se debería escoger la solución que aísla uno de los núcleos ya que nos mejora la latencia máxima y además evita que cualquier aplicación del sistema con prioridades de tiempo real sea asignada y ejecutada en el núcleo aislado.

Resultados de la caracterización de las soluciones Raspbian

Finalmente, para concluir con las soluciones Raspbian, se muestran las tablas de las funciones que se indicó que se iban a medir en el capítulo anterior.

Caracterización de Mutex en Raspbian				
	Raspbian	Raspbian Aislado	Raspbian con RCU	Raspbian Aislado con RCU
Bloqueo de un mutex (*lock*)	Mínimo: 156ns Medio: 230ns Máximo: 437389ns	Mínimo: 156ns Medio: 188ns Máximo: 34896ns	Mínimo: 104ns Medio: 161ns Máximo: 66250ns	Mínimo: 104ns Medio: 161ns Máximo: 68282ns
Bloqueo de un mutex (*trylock*)	Mínimo: 104ns Medio: 181ns Máximo: 562230ns	Mínimo: 104ns Medio: 152ns Máximo: 48282ns	Mínimo: 104ns Medio: 150ns Máximo: 213853ns	Mínimo: 104ns Medio: 139ns Máximo: 74219ns
Desbloqueo de un mutex (*unlock*)	Mínimo: 156ns Medio: 247ns Máximo: 527127ns	Mínimo: 156ns Medio: 213ns Máximo: 48178ns	Mínimo: 156ns Medio: 195ns Máximo: 90469ns	Mínimo: 156ns Medio: 192ns Máximo: 87397ns

Tabla 1. Caracterización de los mutex en Raspbian.

Analizando los datos de la Tabla 1, se puede ver que todas las soluciones mejoran los resultados de los *mutex* respecto a Raspbian, pero hay que hacer hincapié en que no ocurre como en la latencia donde si combinamos las soluciones los resultados mejoraban. Esto se debe a que las soluciones que utilizan RCU, aunque mejoran el tiempo medio, tienen un tiempo máximo muy superior, cerca del doble que la solución con aislamiento. Es por eso por lo que solo aplicar el aislamiento parece una mejor solución en el caso de los *mutex*.

Caracterización de Cambios de Prioridad en Raspbian				
	Raspbian	Raspbian Aislado	Raspbian con RCU	Raspbian Aislado con RCU
Cambio de prioridad (*pthread_ setschedparam*)	Mínimo: 2291ns Medio:2453ns Máximo:222654ns	Mínimo: 2187ns Medio: 2367ns Máximo:309530ns	Mínimo: 3020ns Medio: 3354ns Máximo: 286613	Mínimo: 2864ns Medio: 3111ns Máximo:233228ns
Cambio de prioridad (*pthread_ setschedprio*)	Mínimo:2291ns Medio: 2447ns Máximo:206612ns	Mínimo:2187ns Medio: 2335ns Máximo:110729ns	Mínimo: 2969ns Medio: 3199ns Máximo:179895ns	Mínimo:2865ns Medio:3039ns Máximo:226145ns

Tabla 2. Caracterización de los cambios de prioridad en Raspbian

En el caso de los cambios de prioridad, apenas se ve mejora en ninguna de las soluciones, por ejemplo, en el caso del aislamiento sí que mejora los tiempos medios, y el tiempo máximo es mejor para *pthread_setschedprio* pero es peor para *pthread_setschedparam*. Y para las soluciones donde se incluye RCU los tiempos son bastante peor en todos los casos que para Raspbian normal.

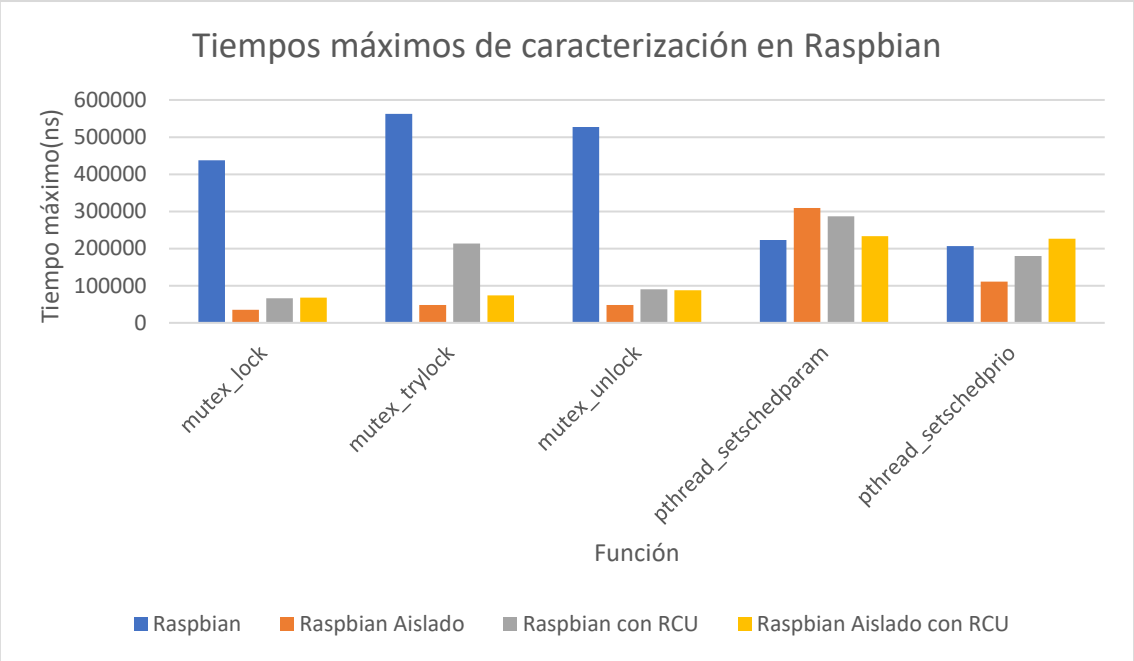

Figura 12.Comparación tiempos máximos de caracterización en Raspbian

Observando la Figura 12 que recoge los tiempos de caracterización de *mutex* y de cambios de prioridad expuestos en la Tabla 1 y Tabla 2 (no se han incluido los datos del *timer* porque son de diferente orden de magnitud) se puede comprobar visualmente la mejora que supone el aislamiento en los tiempos máximos de todas las funciones excepto en la de *pthread_setschedparam* donde el tiempo de Raspbian Aislado es peor que el de Raspbian. Por otro lado, el mecanismo RCU empeora los tiempos máximos en los cambios de prioridad.

Caracterización de Timers en Raspbian				
	Raspbian	Raspbian Aislado	Raspbian con RCU	Raspbian Aislado con RCU
Timer Relativo 1000us (*clock_nanosleep*)	Mínimo: 1059,6us Medio: 1333,7us Máximo:40366,1us	Mínimo:1059,5us Medio: 1103,6us Máximo:2634,1us	Mínimo: 1053,4us Medio: 1340,6us Máximo:22812us	Mínimo:1059,6us Medio: 1110,5us Máximo:1433,6us
Timer Absoluto 1000us (*clock_nanosleep*)	Mínimo: 50,3us Medio: 1382,9us Máximo:77169,1us	Mínimo:1057us Medio: 1336,5us Máximo:1433,9us	Mínimo: 938,3us Medio: 1351us Máximo:38072,2us	Mínimo:1030,1us Medio: 1113,1us Máximo: 1339us

Tabla 3. Caracterización de los timers en Raspbian.

Y los últimos resultados de la caracterización son los del *timer*. Estos datos se han decidido presentar en microsegundos en vez de en nanosegundos para acortar su longitud y la tabla no quedara un tanto ilegible. Aplicando el aislamiento se consigue mejorar drásticamente los resultados, por el contrario, esto no se ve tan reflejado en los resultados de RCU cuyos tiempos máximos sobrepasan los 20000us. Pero al combinar ambas, se consigue reducir los tiempos medios y máximos de los *timers*.

Una vez terminado el análisis del impacto que tienen las soluciones del aislamiento y de RCU sobre *Raspbian*, pasaremos a analizar la solución del parche PREEMP_RT y averiguar si como en el caso anterior, añadiendo las configuraciones del aislamiento y RCU mejorarán los resultados. Primero comenzaremos analizando la mejora que supone *Raspbian* con el parche aplicado frente a no tener parche.

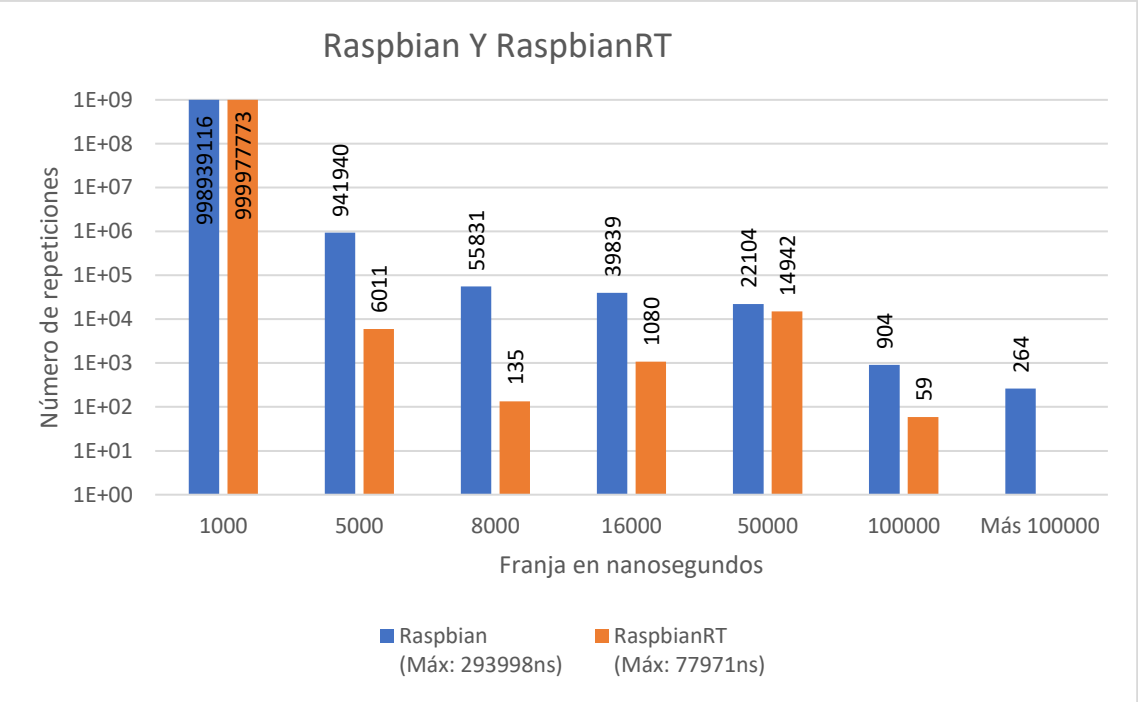

Figura 13. Comparación de Raspbian y RaspbianRT en escala logarítmica.

En primer lugar, se observa que simplemente con la aplicación del parche ya se consigue que ninguna latencia sobrepase los 100000ns y que la latencia máxima sea más baja que en cualquiera de los casos anteriores. Por otro lado, consigue que el 99,99% de los resultados estén en la primera franja que es algo que ya habíamos visto en Raspbian Aislado. Una vez creada la imagen de los beneficios que supone aplicar el parche PREEMP_RT se va a analizar si aplicar las soluciones de Aislamiento o RCU supone beneficios sustanciosos. Como se ha hecho con Raspbian normal, primero se analizará el impacto de aplicar el aislamiento sobre RaspbianRT, después el impacto de añadir las opciones RCU sobre el kernel de RaspbianRT y finalmente se hará una comparación de estas soluciones y su combinación.

Evaluación de RaspbianRT Aislado frente a RaspbianRT

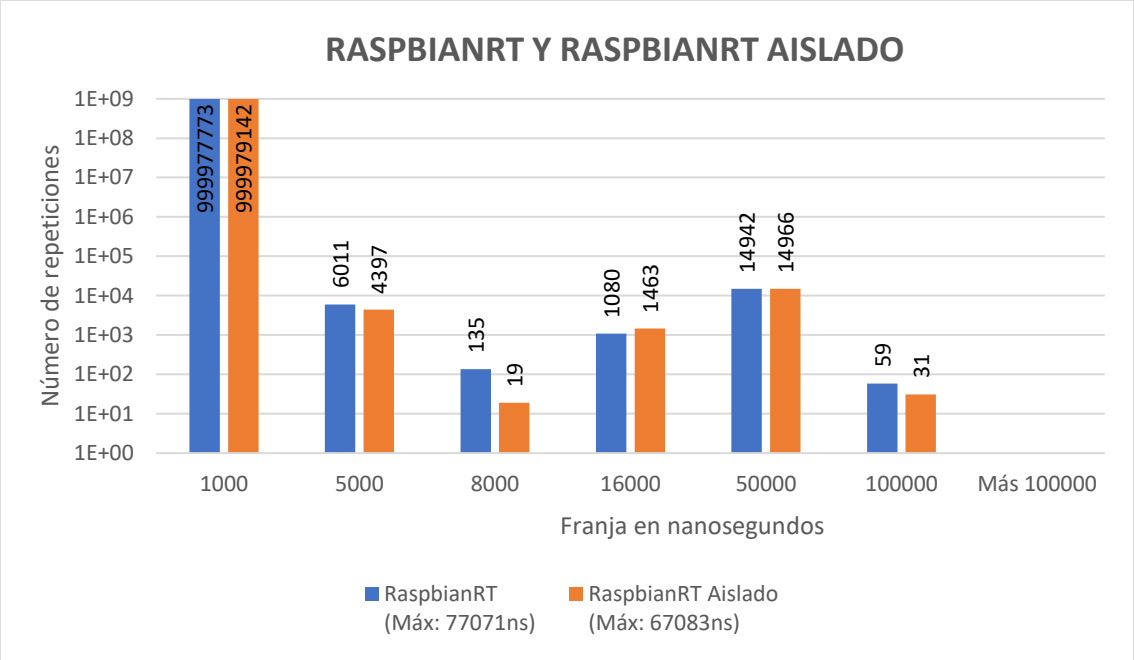

Figura 14. Comparación de RaspbianRT y Raspbian Aislado en escala logarítmica.

En este caso se aprecia poca mejora con la aplicación del aislamiento, dado que solo se reduce 10000ns la latencia máxima y el resto de resultados prácticamente similares, esto puede deberse a que con la configuración del parche se consigue que las aplicaciones de tiempo real no se vean interferidas por el resto del sistema por lo que el aislamiento no consigue mejora alguna, u otro indicio de esta leve mejora se puede asociar a que se está acercando a latencias mucho más bajas que en los casos anteriores por lo que es más difícil alcanzar mejoras tan grandes.

Evaluación de RaspbianRT con RCU frente a RaspbianRT

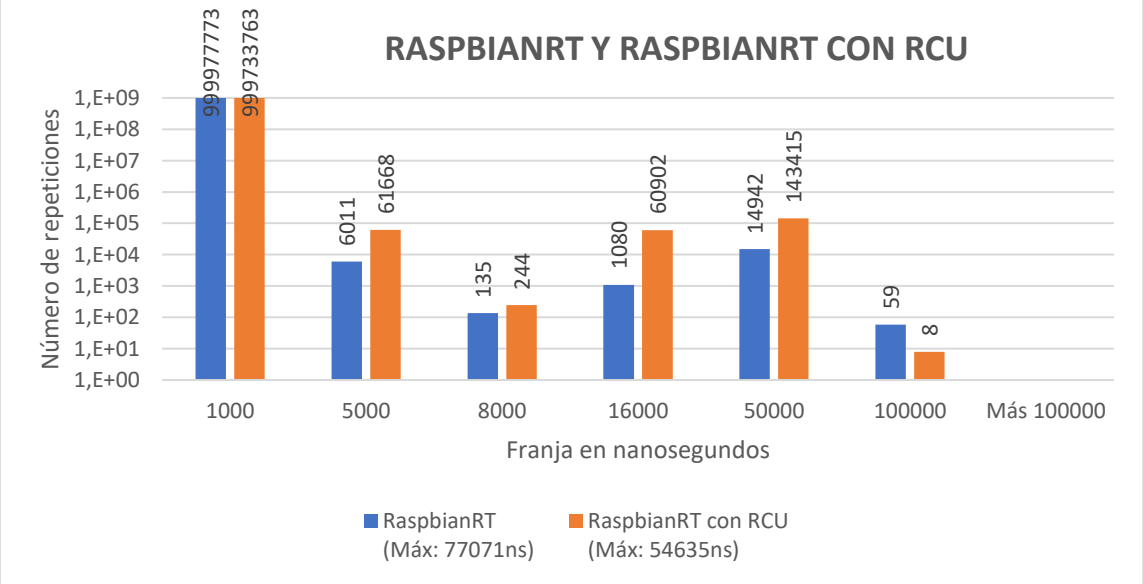

Figura 15. Comparación de RaspbianRT y RaspbianRT con RCU.

En este caso, es difícil dar un veredicto de si el mecanismo RCU realmente mejora o no RaspbianRT. Si se observa la Figura 14, vemos como en la franja de los 100000ns, la solución con RCU tiene un menor número de resultados y la latencia máxima es menor, siendo esta 54635ns. Pero, por otro lado, el porcentaje de la primera franja es peor, 99,99% RaspbianRT y 99,97% RaspbianRT con RCU. Este 0,01% de diferencia causa que en el resto de las latencias RaspbianRT con RCU tenga peores números.

Resumen de latencias para las soluciones RaspbianRT

En la gráfica de la Figura 15, se observa que en las latencias de 5000 a 50000 las soluciones con RCU tienden a tener peores resultados que las soluciones sin RCU. Pero en las latencias por encima de 50000ns, cuando se aplica RCU los resultados son mejores, llegando incluso la combinación de aislamiento y RCU a no sobrepasarlos, con una latencia máxima de 45833ns. Así que, al conseguir una latencia máxima tan baja, se puede afirmar que RaspbianRT con RCU es la mejor solución que se ha estudiado para las aplicaciones de tiempo real, aunque cabe resaltar que no llega a acumular el 99,99% de resultados por debajo de los 1000ns, aunque esto si lo consiguen otras soluciones como RaspbianRT y RaspbianRT Aislado

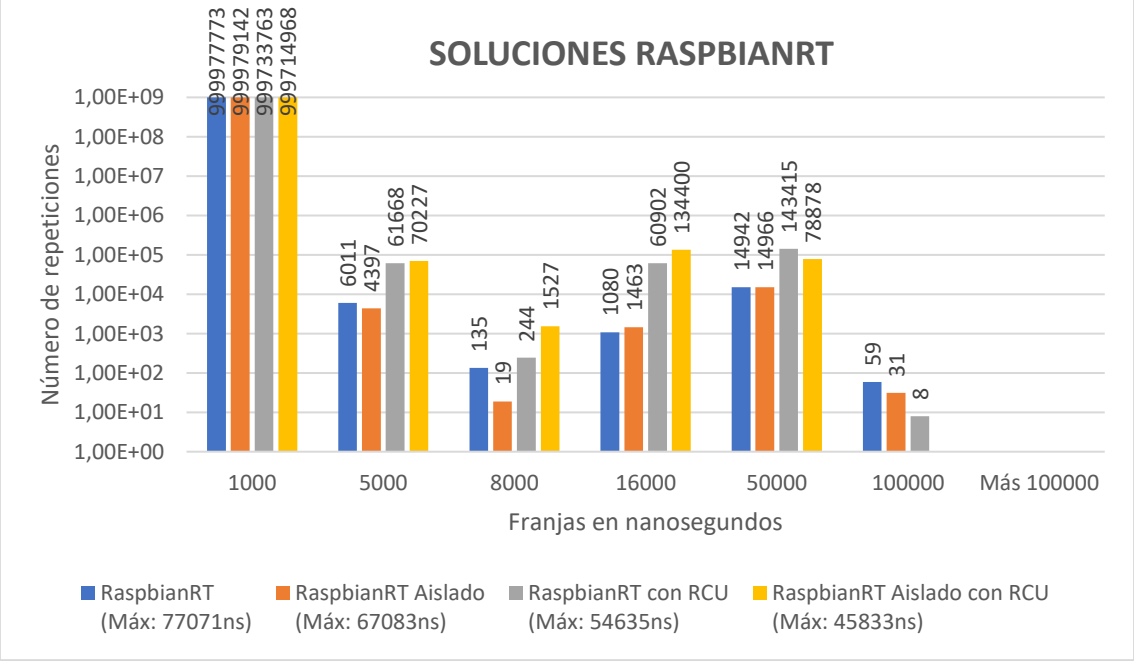

Figura 16. Comparación soluciones RaspbianRT.

Resultados de la caracterización de las soluciones RaspbianRT

Para acabar con las pruebas realizadas sobre este último grupo de soluciones, se presentarán los resultados obtenidos en los test de caracterización de los cuales se mostrará el tiempo mínimo, medio y máximo de cada una de las soluciones para bloqueos y desbloqueos de un *mutex*, los cambios de prioridad y *timers*.

Caracterización de Mutex en RaspbianRT				
	RaspbianRT	RaspbianRT Aislado	RaspbianRT con RCU	RaspbianRT Aislado con RCU
Bloqueo de un *mutex* (*lock*)	Mínimo: 104ns Medio: 256ns Máximo:72763ns	Mínimo: 104ns Medio: 156ns Máximo: 62084ns	Mínimo: 104ns Medio: 254ns Máximo: 65365ns	Mínimo: 104ns Medio: 161ns Máximo: 42136ns
Bloqueo de un *mutex* (*trylock*)	Mínimo: 104ns Medio: 232ns Máximo:79847ns	Mínimo: 104ns Medio: 138ns Máximo:58906ns	Mínimo: 104ns Medio: 230ns Máximo: 72657ns	Mínimo: 104ns Medio: 142ns Máximo: 40677ns
Desbloqueo de un *mutex* (*unlock*)	Mínimo: 156ns Medio: 290ns Máximo:98180ns	Mínimo: 156ns Medio: 186ns Máximo: 42292ns	Mínimo: 156ns Medio: 288ns Máximo: 60522ns	Mínimo: 156ns Medio: 189ns Máximo: 49532

Tabla 4. Caracterización de mutex en RaspbianRT

En el caso de los *mutex*, las mejores soluciones son donde se aplica el aislamiento, ya sea en RaspbianRT Aislado o RaspbianRT Aislado con RCU, en este último los tiempos medios son más altos, pero por el contrario las latencias máximas son más bajas. A pesar de que las soluciones mejoran los tiempos máximos, si observamos la Figura 17, las mejoras no son tan sustanciales como en los cambios de prioridad.

Caracterización de Cambios de Prioridad en RaspbianRT				
	RaspbianRT	RaspbianRT Aislado	RaspbianRT con RCU	RaspbianRT Aislado con RCU
Cambio de prioridad (*pthread_setschedparam*)	Mínimo: 3020ns Medio:3425ns Máximo:313419ns	Mínimo: 2865ns Medio:3148ns Máximo:109592ns	Mínimo: 2916ns Medio:3357ns Máximo:175103ns	Mínimo: 2812ns Medio:3093ns Máximo:106302ns
Cambio de prioridad (*pthread_setschedprio*)	Mínimo: 3020ns Medio:3449ns Máximo:155005ns	Mínimo: 2865ns Medio:3144ns Máximo:87038ns	Mínimo: 2916ns Medio:3358ns Máximo:195415ns	Mínimo: 2812ns Medio:3085ns Máximo:125781ns

Tabla 5. Caracterización cambios de prioridad em RaspbianRT

En cuanto a los tiempos de los cambios de prioridad, se puede decir que el aislamiento es la mejor de las soluciones porque consigue un tiempo máximo en los dos casos, aunque en las opciones donde se aplica RCU también se mejoran los tiempos de RaspbianRT.

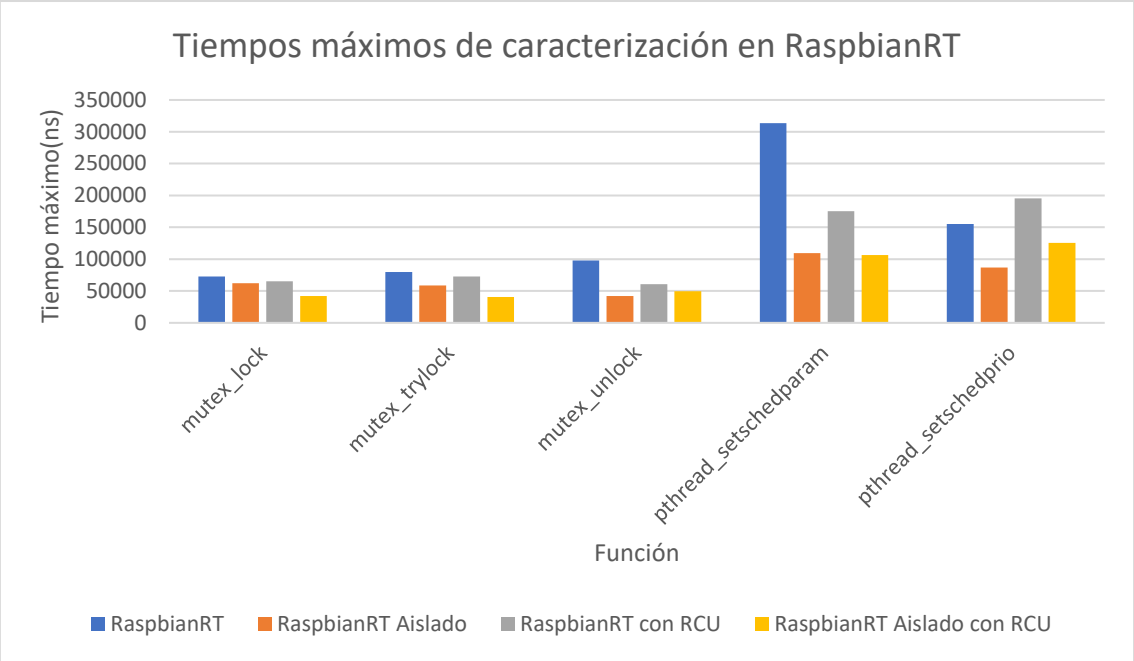

Figura 17. Comparación tiempos máximos de caracterización en RaspbianRT

Como se puede ver en la Figura 17, en general los mecanismos de aislamiento y RCU mejoran los tiempos de RaspbianRT, pero cabe destacar que RaspbianRT es el que ha obtenido tiempos máximos más bajos. Si comparamos esta gráfica con la de la Figura 12, podemos ver que en este caso hay una diferencia importante en los cambios de prioridad mientras que en la Figura 12 la mejora sustancial se conseguía en los *mutex.*

Caracterización de Timers en RaspbianRT				
	RaspbianRT	RaspbianRT Aislado	RaspbianRT con RCU	RaspbianRT Aislado con RCU
Timer Relativo 1000us (*clock_nanosleep*)	Mínimo:1029,3us Medio: 1168,9us Máximo:29714,3us	Mínimo:1069,3us Medio: 1348,7us Máximo:1226,5us	Mínimo:1050,1us Medio: 1088,9us Máximo:9525,7us	Mínimo:1072,2us Medio: 1096us Máximo:3195,1us
Timer Absoluto 1000us (*clock_nanosleep*)	Mínimo: 1050,8us Medio:1088,5us Máximo:24086,3us	Mínimo:1067,9us Medio: 1093,1us Máximo:1300,4us	Mínimo: 978,8us Medio: 1137us Máximo:16011,1us	Mínimo:1023,8us Medio: 1089,1us Máximo:2618,3us

Tabla 6. Caracterización de timers en RaspbianRT

En el caso de los *timers* ocurre como en los *mutex* y los cambios de prioridad, las soluciones donde se ha aplicado el aislamiento tienen tiempos máximos más bajos que el resto soluciones. Por otro lado, la solución RaspbianRT con RCU no produce una mejora tan sustancial, como las otras, siendo en algunos casos incluso peor que RaspbianRT como en el cambio de prioridad (*pthread_setschedprio*)

4.2. Evaluación de las soluciones Android

En esta sección realizaremos las comparaciones de las soluciones que se han probado en Android y si estas mejoran el rendimiento. Como se introdujo en el inicio del capítulo, en Android solo se aplicarán dos soluciones el aislamiento y la solución RTAndroid. Para mostrar los resultados de Android se seguirá una estructura similar a la desarrollada en la sección anterior de Linux. Primero se establecerá una línea base de los resultados que puede conseguir Android, y se comparará las mejoras que aportan el resto de las soluciones. Para las pruebas de Android se ha realizado la carga de trabajo del sistema ejecutando la aplicación Antutu Benchmark, que realiza un test de estrés para ver cómo se comporta el dispositivo con grandes cargas de trabajo y darle una puntuación con la cual compararlo con el resto de los dispositivos.

Evaluación Android

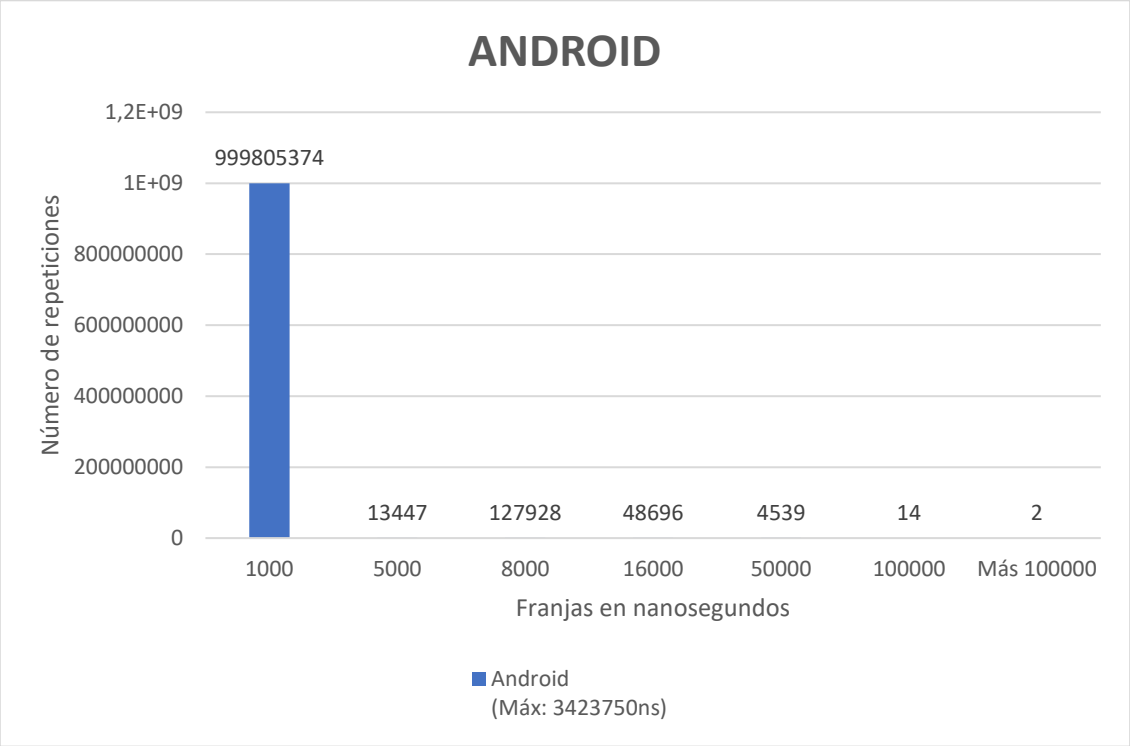

Figura 18. Tiempos de latencia en Android.

Cómo se puede observar en la gráfica de la Figura 16, el 99,98% de los resultados se encuentran en la primera franja, es decir, por debajo de los 1000ms. Como en Linux, casi todos los resultados se encuentran en esta franja y solo una cantidad muy pequeña en comparación con el número total de resultados se desplaza al resto de franjas. Aunque en esta línea base solo hay dos resultados por encima de los 100000ns, hay un valor que se dispara por encima del resto llegando hasta los 3423750ns la cual es una cifra demasiado alta y seguramente haría que una aplicación de tiempo real sobrepasase su *deadline*. Es por eso por lo que hay que estudiar si las soluciones disponibles mejoran esta latencia máxima.

Evaluación de Android Aislado frente a Android

Como en otras ocasiones, cuando se aplica el aislamiento se mejoran los resultados de las franjas más altas y además incluso se consigue que las latencias sean menores de 100000ms, que como se ha indicado cuando se analizaba los resultados de Linux, este aspecto es importante de cara a los plazos de las aplicaciones en tiempo real. Ahora se pasará a analizar los resultados obtenidos en RTAndroid y comparar su mejora frente a Android stock.

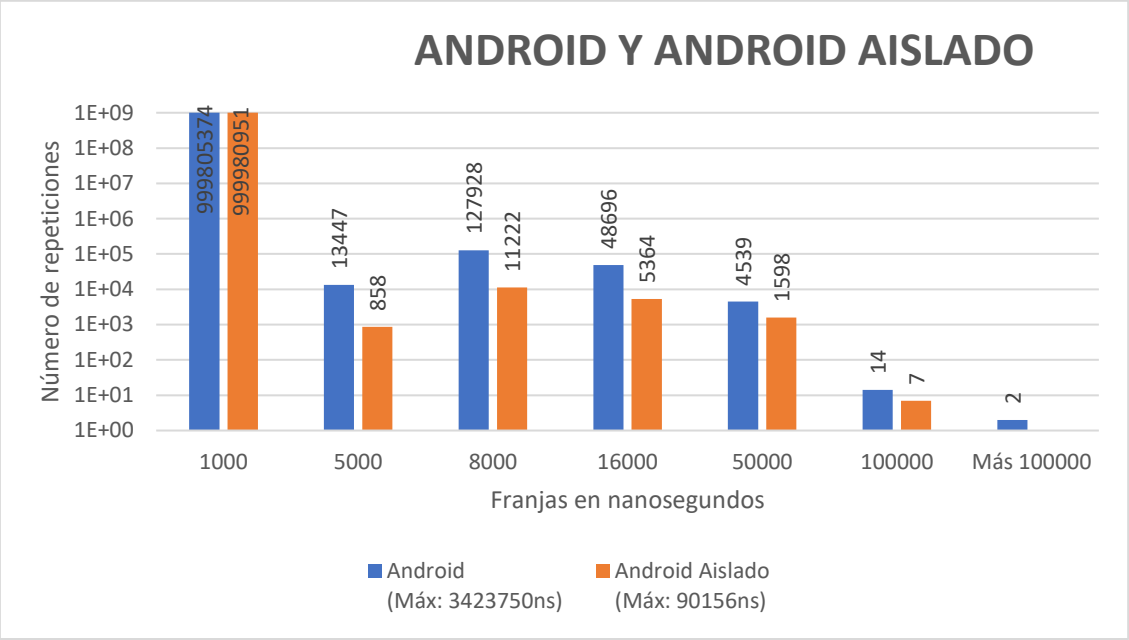

Figura 19. Comparación de Android y Android Aislado.

Evaluación de RTAndroid frente a Android

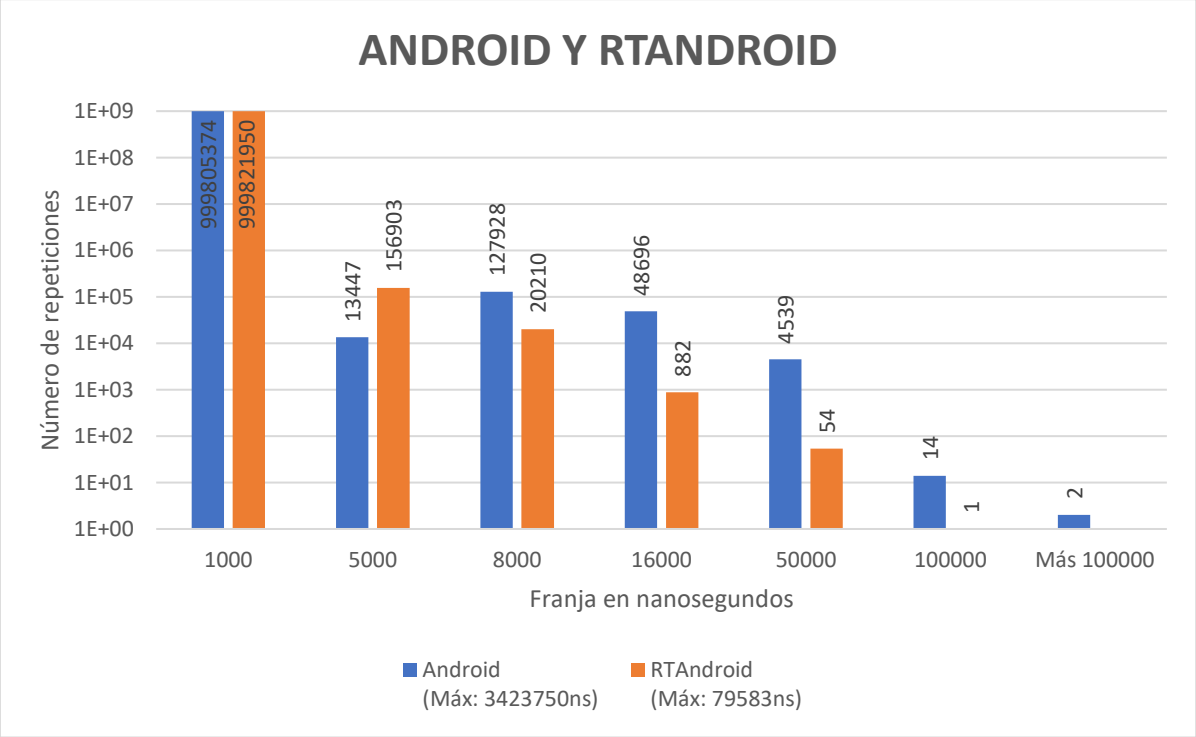

Figura 20. Comparación de Android y RTAndroid

En esta comparación se puede ver como en RTAndroid crece el número de latencias por debajo de los 5000ns, y que es más bajo en los resultados que sobrepasan los 5000ns. Tanto en esta solución como con el aislamiento la latencia máxima se ha reducido enormemente en comparación con los 3 milisegundos que se han obtenido en Android base. En este caso la latencia máxima es de 79583ns y es la única medida que sobrepasa los 50000ns en RTAndroid.

Resumen de latencias para las soluciones Android

A forma de resumen, y como se hizo en Raspbian se va a mostrar una gráfica en la que se comparan las dos soluciones presentadas para Android y la combinación de ambas.

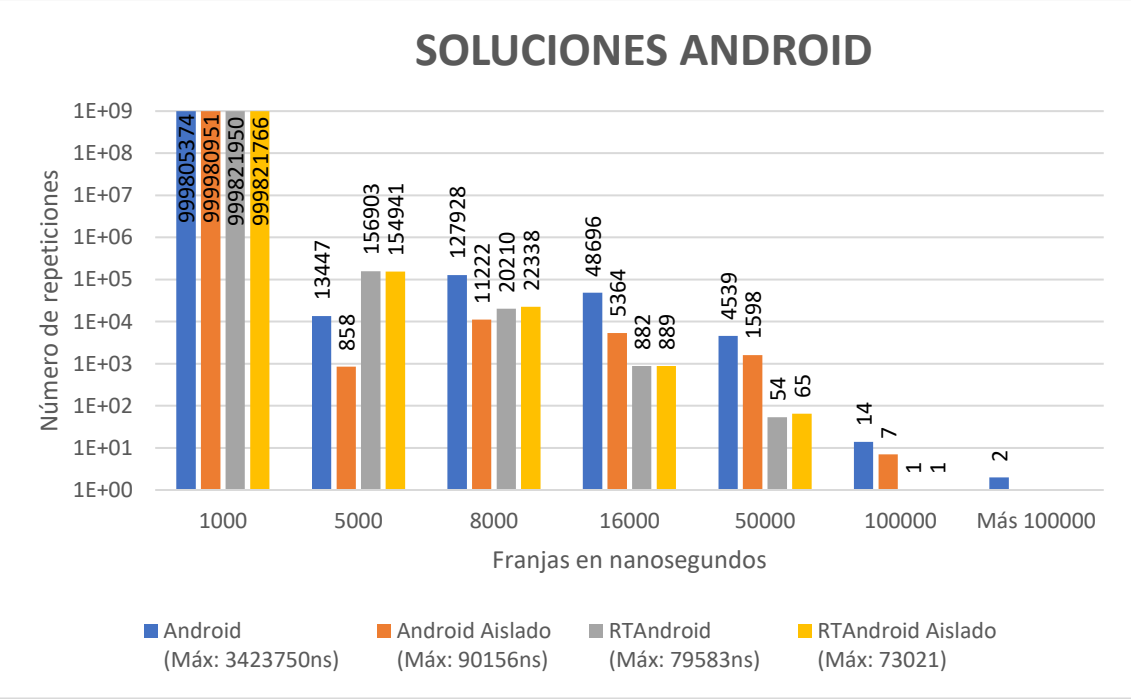

Figura 21. Comparación de las soluciones Android

Como conclusión de la gráfica podemos decir que casi no hay diferencia entre las soluciones RTAndroid y RTAndroid Aislado, aunque este último presenta una latencia máxima más baja. Pero sí que hay una diferencia notable entre las soluciones RTAndroid y Android normal, donde las latencias máximas son más altas en estas últimas.

Resultados de la caracterización de las soluciones Android

En esta sección se mostrarán los últimos resultados relacionados con Android que son los resultados de caracterización, que como en los casos anteriores incluyen *mutex, timers* y cambios de prioridad.

En la Tabla 7, se observa como todas las soluciones mejoran notablemente los tiempos máximos de los *timers*, aunque los mejores tiempos son conseguidos por la combinación de ambas soluciones, es decir RTAndroid Aislado.

Caracterización de Mutex en Android				
	Android	Android Aislado	RTAndroid	RTAndroid Aislado
Bloqueo de un mutex (*lock*)	Mínimo: 52ns Medio: 98ns Máximo: 68021ns	Mínimo: 52ns Medio: 104ns Máximo: 17761ns	Mínimo: 52ns Medio: 90ns Máximo: 14062ns	Mínimo: 52ns Medio: 89ns Máximo: 8750ns
Bloqueo de un mutex (*trylock*)	Mínimo: 52ns Medio:92ns Máximo: 53802ns	Mínimo: 52ns Medio: 90ns Máximo:49479ns	Mínimo: 52ns Medio: 82ns Máximo: 11614ns	Mínimo: 52ns Medio: 81ns Máximo: 10938ns
Desbloqueo de un mutex (*unlock*)	Mínimo: 104ns Medio: 165ns Máximo:141302ns	Mínimo: 104ns Medio: 135ns Máximo: 33646ns	Mínimo: 104ns Medio: 118ns Máximo: 23073ns	Mínimo: 104ns Medio: 118ns Máximo: 11146ns

Tabla 7. Caracterización de mutex en Android.

Como en el caso de los *mutex*, en los cambios de prioridad también se perciben mejoras a medida que se combinas las opciones, reduciéndose el tiempo máximo en algunos casos hasta 100000ns si comparamos los resultados de *pthread_setschedprio* de Android y RTAndroid Aislado.

Caracterización de Cambios de Prioridad en Android				
	Android	Android Aislado	RTAndroid	RTAndroid Aislado
Cambio de prioridad (*pthread_setschedparam*)	Mínimo: 3646ns Medio:3949ns Máximo: 137448	Mínimo: 3698ns Medio:3939ns Máximo:166354ns	Mínimo: 2812ns Medio:3064ns Máximo: 88802ns	Mínimo: 2916ns Medio:3153ns Máximo:53230ns
Cambio de prioridad (*pthread_setschedprio*)	Mínimo: 3698ns Medio:3935ns Máximo:141667ns	Mínimo: 3698ns Medio: 3941ns Máximo:106198ns	Mínimo: 2812ns Medio:2965ns Máximo: 84219ns	Mínimo: 2864ns Medio:2993ns Máximo:37656ns

Tabla 8. Caracterización de cambios de prioridad en Android.

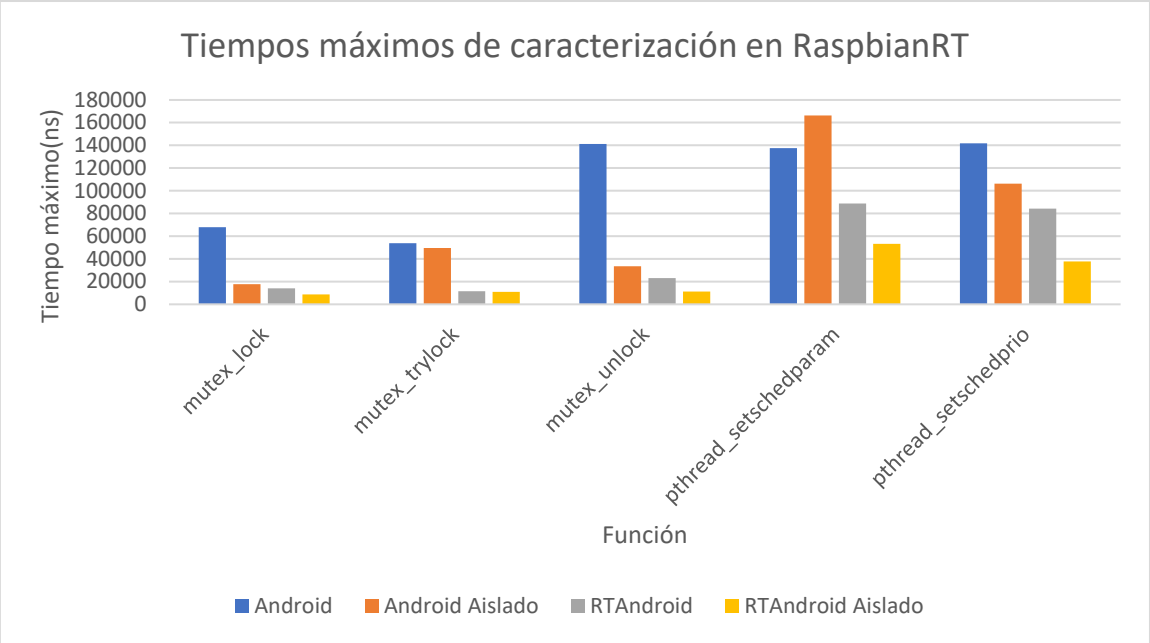

Figura 22. Comparación de tiempos máximos de caracterización en Android

En la Figura 22, se puede comparar de una forma visual los tiempos máximos de las funciones en Android, y se observa que el aislamiento mejora los tiempos en todos los casos menos en *pthread_setschedparam* pero donde realmente se mejora el comportamiento temporal de la caracterización es cuando se utiliza RTAndroid, que tiene tiempos máximos más bajos tanto si usa aislamiento como si no.

Y finalmente, en los *timers* no se percibe la mejoría de resultados a medida que aplicamos más soluciones a Android, en realidad lo que se produce es un efecto negativo siendo RTAndroid la peor solución de todas, y empeorando los tiempos de Android base incluso el doble.

Caracterización de Timers en Android				
	Android	Android Aislado	RTAndroid	RTAndroid Aislado
Timer Relativo 1000us (*clock_nanosleep*)	Mínimo: 1059,6us Medio:1075us Máximo: 2842,2us	Mínimo: 1059,4us Medio:1066us Máximo: 1192,4us	Mínimo: 1015,1us Medio: 1063,5us Máximo: 2169us	Mínimo: 1012,1us Medio: 1077,7us Máximo: 4929,8us
Timer Absoluto 1000us (*clock_nanosleep*)	Mínimo: 838,8us Medio:1066,7us Máximo:1334,3us	Mínimo: 1058,5us Medio: 1063,4us Máximo: 1126,7us	Mínimo: 977,3us Medio: 1066,6us Máximo: 2492,8us	Mínimo: 1014,8us Medio: 1086,2us Máximo: 4647,9us

Tabla 9. Caracterización de timers en Android

4.3. Interferencias de otras tareas de tiempo real

Cómo última prueba, se ha querido demostrar que con el aislamiento se genera un plus de seguridad, aunque en algunos casos no tenga mejorías notables. Para ello se ha diseñado un programa para cargar el sistema que lanzará tres hilos con prioridades baja,

media y alta por cada uno de los núcleos disponibles en el dispositivo. Cada uno de esos hilos ocupa el procesador durante 17000us aproximadamente y después se duerme durante 100000us. Con esto se consigue que si se lanza el Cyclictest a una prioridad media siempre habrá un hilo que tenga más prioridad que este último. Excepto obviamente cuando se aplique la solución del aislamiento que solo podrán ejecutarse en el CPU aislado las aplicaciones cuyo PID este en el fichero correspondiente.

Kernel del Sistema Operativo	Tiempo Medio (us)	Tiempo Máximo (us)
Raspbian	149	16256
Raspbian Aislado	71	9550
Raspbian RT	100	16956
Raspbian RT Aislado	9	54

Tabla 10. Tiempos del peor caso en Raspbian.

Para recoger las medidas se utilizará el programa Cyclictest, que se lanzará con prioridad 50 de la forma que se explicó en el capítulo anterior. Cómo se observa en la Tabla 10, en las opciones en las que no se utiliza el aislamiento el tiempo de latencia máxima es casi el tiempo que se ejecuta cada uno de los hilos antes de dormirse. Esto se debe porque al tener una aplicación de mayor prioridad es obvio que el planificador le concede antes el uso de CPU y en el peor de los casos si ambos lo piden al mismo tiempo el hilo de baja prioridad tendrá que esperar hasta que el hilo de mayor prioridad acabe, es decir los 17000us que se ejecutan los hilos del programa de estrés. Por lo que en este experimento se demuestra que si aplicamos el mecanismo de aislamiento nos aseguramos de que otras aplicaciones de tiempo real no interfieran con nuestro programa.

5. Conclusiones y Trabajos futuros

En el presente trabajo se han analizado diferentes alternativas para mejorar el comportamiento temporal de Linux/Android en las aplicaciones de tiempo real. Para Linux se ha considerado aplicar el parche PREEMPT_RT y el mecanismo de aislamiento [1] propuesto por el grupo ISTR de la Universidad de Cantabria. Primero se han realizado pruebas sin simular interferencias con otras aplicaciones de tiempo real y se han recogido las medidas para todas las combinaciones de soluciones posibles. Con la obtención de estos resultados se ha observado que las alternativas de Linux que se han estudiado, tanto el parche PREEMPT_RT como el mecanismo de aislamiento, mejoran el comportamiento temporal de este, y que la combinación de ellas entre si hace que los resultados sean mejores. Por otro lado, se ha comparado la solución propuesta por el grupo ISTR [1][5] para mejorar las capacidades de Android para soportar aplicaciones de tiempo real y la solución RTAndroid [6][7], este caso tampoco se ha simulado que otras aplicaciones con prioridades de tiempo real estaban estresando el sistema. Tras observar los resultados se ha podido concluir que la alternativa RTAndroid mejora los tiempos de latencia pero que añadiéndole el mecanismo de aislamiento obtiene unos beneficios mayores. Finalmente, se ha hecho un experimento para demostrar que ocurre cuando mientras se hacen las pruebas hay interferencias con otras aplicaciones de tiempo real con una prioridad mayor. Los resultados han mostrado que cuando utilizamos el mecanismo de aislamiento también se evitan las interferencias con otras aplicaciones de tiempo real y por tanto es recomendable usarlo si queremos que una aplicación de tiempo real se ejecute correctamente.

Como futuros trabajos se plantean las siguientes opciones:

- Repetir las pruebas realizadas en este trabajo, pero adaptándolas a Java y hacer comparaciones con otras máquinas virtuales de tiempo real como Fiji. Y evaluar la solución propuesta por RTAndroid [6][7] para poder ejecutar aplicaciones Java con requisitos temporales sobre la plataforma Android.

- Realizar un demostrador para verificar que, en un entorno real, las soluciones que se han estudiado obtienen unos resultados parecidos a los expuestos en este trabajo.

Bibliografía

[1] Alejandro Pérez Ruiz, Mario Aldea Rivas, and Michael González Harbour. 2015. CPU Isolation on the Android OS for running Real-Time Applications. In Proceedings of the 13th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES '15). ACM, New York, NY, USA, Article 6, 7 pages. DOI: https://doi.org/10.1145/2822304.2822317

[2] **Parche PREEMPT_RT**
https://rt.wiki.kernel.org/index.php/Frequently_Asked_Questions
Visitada por última vez el 20/06/2019.

[3] **Earliest Deadline First**
https://en.wikipedia.org/wiki/Earliest_deadline_first_scheduling
Visitada por última vez el 20/06/2019.

[4] **Inversión de Prioridades**
https://wiki.linuxfoundation.org/realtime/documentation/technical_basics/pi
Visitada por última vez el 20/06/2019

[5] Alejandro Pérez Ruiz, Mario Aldea Rivas, and Michael González Harbour. 2016. Servicios de tiempo real en Android. V Simposio de Sistemas de Tiempo Real in the V Congreso Español de Informática, CEDI 2016, Salamanca (Spain), September 2016.

[6] Igor Kalkov, Dominik Franke, John F. Schommer, and Stefan Kowalewski. 2012. A real-time extension to the Android platform. In Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES '12). ACM, New York, NY, USA, 105-114. DOI: http://dx.doi.org/10.1145/2388936.2388955

[7] Igor Kalkov, Alexandru Gurghian, and Stefan Kowalewski. 2015. Priority Inheritance during Remote Procedure Calls in Real-Time Android using Extended Binder Framework. In Proceedings of the 13th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES '15). ACM, New York, NY, USA, Article 5, 10 pages. DOI: https://doi.org/10.1145/2822304.2822311

[8] Yin Yan, Shaun Cosgrove, Varun Anand, Amit Kulkarni, Sree Harsha Konduri, Steven Y. Ko, and Lukasz Ziarek. 2014. Real-time android with RTDroid. In Proceedings of the 12th annual international conference on Mobile systems, applications, and services (MobiSys '14). ACM, New York, NY, USA, 273-286. DOI: http://dx.doi.org/10.1145/2594368.2594381

[9] **Evaluación del sistema operativo Android para aplicaciones de tiempo real.**
Santiago Sañudo Martinez