Universidad de Zaragoza

Sistemas Empotrados 2

Trabajo de asignatura

Medida de latencias en Linux usando Ftrace

Autor:

Adrián Martín Marcos 756524

Zaragoza, España

Curso 2020 – 2021

Escuela de
Ingeniería y Arquitectura
Universidad Zaragoza

Resumen

Este trabajo ha consistido en modificar la distribución Linux de la práctica 3, configurando el kernel y añadiendo las herramientas necesarias para medir la latencia de ciertos eventos usando la herramienta Ftrace, y con ello poder hacer una comparativa del rendimiento del sistema sobre la BeagleBone cuando se le somete a diferentes niveles de carga de trabajo.

En este documento se detallan los pasos seguidos y el análisis de los resultados obtenidos en las pruebas realizadas. Los script utilizados y otros ficheros relevantes se incluyen en un directorio llamado *ficheros*.

A lo largo de la memoria se usa terminología y se dan explicaciones que pueden no ser del todo precisas o rigurosas. Algunos de los conceptos que se tratan requieren de un análisis mucho más profundo, que no ha sido posible realizar por los plazos en los que se ha realizado este trabajo. No obstante, y aunque sea de una manera más informal, se han tratado de plasmar los conceptos más importantes que se han aprendido, si bien el documentar todas las horas invertidas en investigación sobre los temas tratados sería imposible.

Índice

1. La herramienta Ftrace

En este apartado se describe de forma breve la herramienta Ftrace, así como algunas de sus características sobre las que se ha aprendido.

También se comentan brevemente algunas de las pruebas que se hicieron en una máquina virtual como primera toma de contacto.

1.1. Qué es y cómo funciona

Ftrace es una herramienta que permite obtener trazas de la ejecución de ciertas actividades en un núcleo Linux. Empezó siendo un tracer independiente para funciones del kernel, pero con el tiempo ha acabado formando parte de él. Además del *function tracer*, cuenta con otras muchas utilidades de tracing, por lo que en la documentación de la última versión de Linux [4] es definido como un framework para tracing en el kernel.

Entre otras cosas, permite hacer tracing estático basado en tracepoints (son macros en el código que hacen log de ciertos eventos, sin interrumpir la ejecución de la función en la que se ejecutan) y tracing dinámico usando kprobes (reemplazan o añaden en ciertos puntos del kernel instrucciones de salto a rutinas en las que se llevan a cabo diferentes acciones de tracing). Con ello se puede depurar el kernel, monitorizar eventos, hacer profiling, medir latencias de ciertas actividades, etc.

Su funcionamiento interno, de forma muy simplificada, se basa en el registro de los eventos de los que se está haciendo tracing en una estructura de datos del kernel llamada *ring buffer*, que opera a modo de cola circular. Toda la interacción con el subsistema de tracing queda expuesta al administrador/usuario del sistema a través de un pseudosistema de ficheros, llamado *tracefs*, que debe ser montado bajo */sys/kernel/debug/tracing*. A través de sus ficheros se puede configurar y consultar la actividad de tracing que se esté realizando en el núcleo.

Tal y como se comentaba, Ftrace dispone de muchas utilidades de tracing, llamadas tracers o plugins, que a partir de ciertos eventos obtienen métricas de los mismos o llevan a cabo algún procesamiento con ellos. Para que estén disponibles en tiempo de ejecución, su código debe haberse compilado como parte del kernel. Éste se encuentra siempre precedido por directivas de compilación condicional, de tal forma si en la configuración del kernel se especifica que ciertos tracers se generen, durante la compilación se definirán las macros necesarias para que el preprocesador de C incluya ese código en el build de Linux.

1.2. Primeras pruebas en una máquina virtual Debian

La primera aproximación que se hizo a Ftrace fue usando una máquina virtual Debian, ya que aunque se disponía de Ubuntu instalado en distintos ordenadores, el miedo a causar daños en el sistema hizo que se optase por la opción de la virtualización.

En ella se realizaron algunas pruebas sencillas (por ejemplo, programas en C que realizaban las llamadas al sistema *write* y *read*), de las cuales se obtuvo una traza usando directamente Ftrace a través del sistema de ficheros.

Fue entonces cuando se vio que esa forma de trabajar con la herramienta resultaba muy tediosa a la hora de realizar pruebas, por lo que desde ese momento se empezó a usar la utilidad *trace-cmd*.

2. Configuración y compilación de Linux

En este apartado se va a detallar el proceso de configuración y compilación del kernel Linux utilizado en prácticas para habilitar aquellos tracepoints del código que se deseaba utilizar para el trabajo.

También se van a comentar brevemente los modelos de expulsión configurados en cada una de las compilaciones realizadas.

2.1. Obtención y preparación de los fuentes

Primero, se han obtenido los fuentes del kernel del directorio de la asignatura en Hendrix [8]. Éstos cuentan ya con el *patch RT* aplicado, por lo que se puede pasar a preparar la compilación directamente:

```
# Descomprimir el kernel
tar xJf kernel-rt.xz

# Preparar fuentes
cd kernel-rt/kernel
make ARCH=arm clean
mkdir rootfs
```

Tras ello, se ejecuta *make menuconfig* para poder llevar a cabo la configuración del kernel mediante un menú *ncurses* en lugar de editar directamente las variables del fichero de texto *.config*:

```
# Abrir menú ncurses para la configuración
make ARCH=arm menuconfig
```

En los siguientes apartados se trata la configuración, usando dicho menú, del modelo de expulsión y los tracers que durante la compilación se quieren incluir.

2.2. Modelo de expulsión

Para elegir el modelo de expulsión del kernel se debe navegar a la sección *Kernel Features >Preemption Model* desde el menú principal. En ella, al haber aplicado el *patch RT*, se pueden ver las opciones siguientes:

Para el trabajo se van a utilizar los modelos *no forced preemption* y *full preemption*.

No forced preemption es el modelo de expulsión original de UNIX, y se utiliza especialmente para servidores. En él los *KCP*s en contexto de interrupción no se pueden bloquear, ya que no son planificables.

Por otra parte, *full preemption* es el modelo de expulsión pensado para tiempo real en Linux. En él todos los *KCP*s, incluidos los que se ejecutan en contexto de interrupción, son planificables. Entre otras modificaciones, aplicarlo supone que todos los *spinlocks* del kernel se convierten en semáforos, por lo que la espera activa en caso de bloqueo se convierte en una invocación directa al planificador. Con esa y otras medidas se busca maximizar el número de expulsiones que se realizan.

Dado que solo se puede configurar un modelo de expulsión, se han realizado dos compilaciones del kernel en las que solo se ha cambiado esta opción, dejando todas las que se comentan a continuación de igual forma.

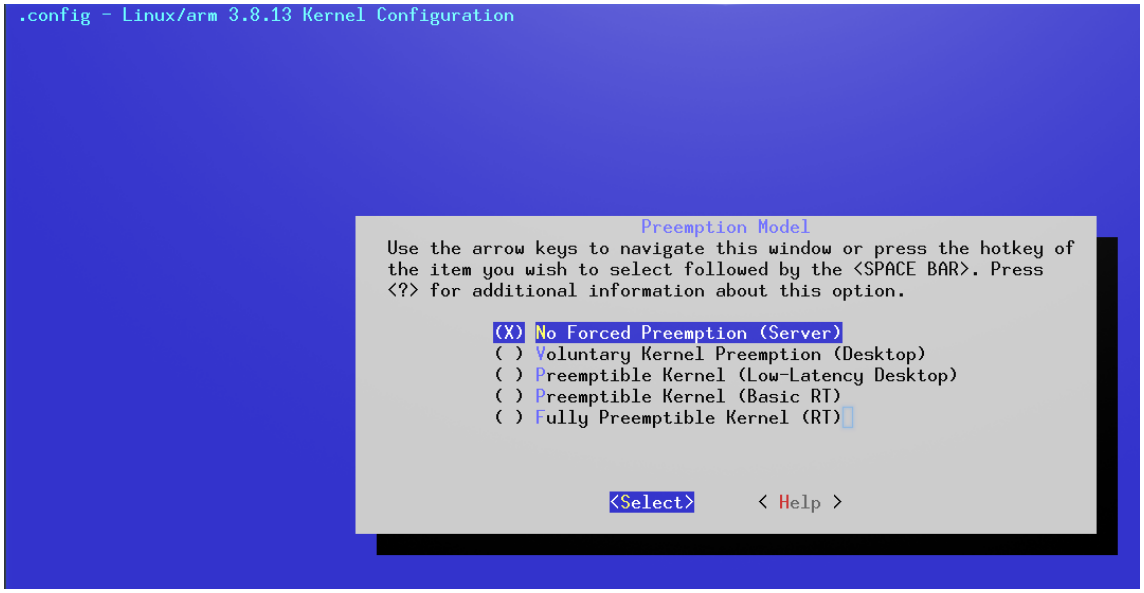

Figura 1: Captura de la configuración establecida en menuconfig

2.3. Tracers

Para elegir llegar a la sección de configuración de los tracers hay que navegar a la sección *Kernel Hacking* > *Tracers* desde el menú principal.

Para la realización del trabajo se han seleccionado los tracers siguientes: *Kernel Function Tracer*, *Interrupts-off Latency Tracer*, *Scheduling Latency Tracer*, *Scheduling Latency Histogram* y *Trace syscalls*. Además, se han dejado marcadas aquellas opciones que se señalaban automáticamente al elegir las mencionadas. Así, las opciones del menú de configuración correspondiente a los tracers quedaba de la siguiente manera:

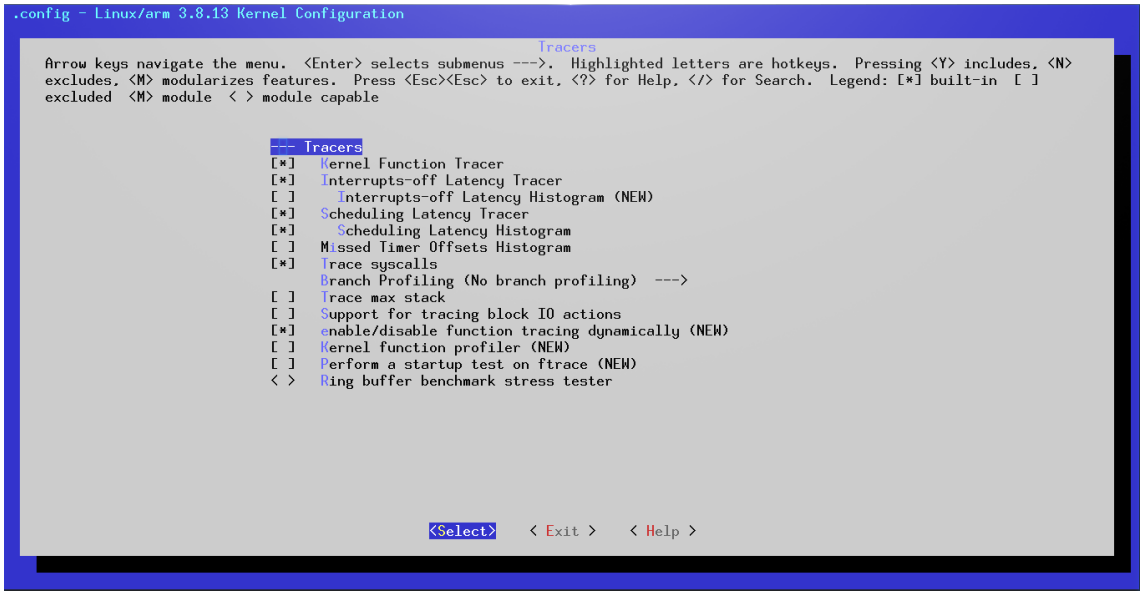

Figura 2: Captura de la configuración establecida en menuconfig

Cabe destacar que la opción de elegir algunos tracers es dependiente del modelo de expulsión que se haya configurado; por ejemplo, el tracer *Preemption-off Latency Tracer* solo aparece cuando se ha configurado antes el modelo de expulsión como *Fully Preemptible Kernel*.

2.4. Compilación

Una vez realizada la configuración deseada, se procede a compilar el kernel, y se hace de manera análoga a como se hizo en la práctica tres de la asignatura:

```
# Compilación del núcleo
export CC="/usr/local/linaro/gcc/bin/arm-linux-gnueabihf-"
make ARCH=arm CROSS_COMPILE=${CC} uImage dtbs
make ARCH=arm CROSS_COMPILE=${CC} modules
make ARCH=arm CROSS_COMPILE=${CC} INSTALL_MOD_PATH=${PWD}/rootfs modules_install
make ARCH=arm CROSS_COMPILE=${CC} uImage-dtb.am335x-bone
```

Este proceso se ha llevado a cabo para las dos configuraciones correspondientes a los distintos modelos de expulsión que se querían probar, dando como resultado dos kernels llamados *zImage_ftrace_FPK* y *zImage_ftrace_NFP*, siendo las letras del final las siglas del modelo de expulsión con el que se configuró la compilación.

3. Herramientas

En este apartado se va a comentar brevemente el proceso de compilación de cada una de las herramientas adicionales que se han instalado en la distribución para facilitar la realización de pruebas con ftrace y la generación de carga de trabajo para las mismas.

Dado que la distribución no cuenta con librerías compartidas instaladas , es necesario que la compilación se realice mediante enlace estático, para que no haya dependencias no satisfechas en tiempo de ejecución.

Por otra parte, y al igual que ha pasado con el kernel y con la herramienta *busybox*, es necesario hacer cross-compiling, ya que la arquitectura target es *ARM*.

Cabe destacar la importancia que tuvieron los ficheros *README* de cada uno de los repositorios correspondientes a las herramientas que se iban a compilar, ya que aparte de éstos no se disponía de más documentación que lo extrapolable del propio fichero *Makefile* para la construcción.

3.1. Trace-cmd

Como ya se comentó en el primer apartado de la memoria, se decició utilizar esta herramienta dada la incomodidad que suponía trabajar directamente con ftrace a través del sistema de ficheros.

El proceso de compilación comenzaba descargando los fuentes y revisando la versión que se deseaba utilizar:

```
git clone git://git.kernel.org/pub/scm/utils/trace-cmd/trace-cmd.git
git checkout trace-cmd-v2.9.1
```

Después, bastaba con dar valor a las variables *LDFLAGS* y *CC* del *Makefile* para indicar, respectivamente, que el enlace debía ser dinámico y cuál software de cross-compilación que se deseaba utilizar:

```
export CC="/usr/local/linaro/gcc/bin/arm-linux-gnueabihf-"
make LDFLAGS=-static CC=${CC}gcc trace-cmd
```

Tras ello, en el directorio *tracecmd/* del repositorio quedaba ubicado el ejecutable correspondiente a la herramienta en cuestión.

3.2. Stress-ng

El uso de *stress-ng* se debió a los problemas que hubo para portar la herramienta *sysbench* a la distribución; todos ellos derivaban de la necesidad de instalar y configurar aparte la librería *LuaJIT*.

Dado que se encontró *stress-ng* y se consiguió portarlo a la distribución en relativamente poco tiempo, fue ésta la herramienta que se utilizó para realizar las pruebas del trabajo.

El proceso de compilación comenzaba descargando los fuentes y revisando la versión que se deseaba utilizar:

```
git clone https://github.com/ColinIanKing/stress-ng.git
git checkout V0.11.00
```

Cabe destacar que en el caso de esta herramienta, se siguieron las instrucciones del *README* para Android [10], ya que era la plataforma documentada más parecida a la que se iba a utilizar (cross-compiling para Linux sobre ARM).

Así, la compilación consistió en ejecutar los comandos siguientes:

```
export $(dpkg-architecture -aarmhf)
export CROSS_COMPILE="/usr/local/linaro/gcc/bin/arm-linux-gnueabihf-"
export CCPREFIX=${CROSS_COMPILE}
export CC=${CROSS_COMPILE}gcc
make ARCH=arm STATIC=1
```

Como resultado, se generó el binario ejecutable correspondiente a la herramienta.

4. Modificaciones en la distribución

En este apartado se van a comentar las modificaciones a la distribución de la práctica tres que se tuvieron que realizar para el trabajo.

4.1. Kernel

Dado que se utilizan dos núcleos en el trabajo, y con el fin de evitar tener que renombrarlos cada vez que se quiera usar uno u otro, se han copiado ambos bajo */boot* con los nombres indicados en el apartado 2 de la memoria y se ha creado un enlace simbólico, llamado *zImage* (pues es el nombre con el que se espera encontrar el kernel bajo */boot*), a aquel con el cuál se quiere arrancar la distribución.

Esto se hace con el comando siguiente:

```
# zImage es solo un enlace simbólico a aquel núcleo con el que se quiere trabajar
ln -s zImage_ftrace_[NFP o FPK] zImage
```

4.2. Script de inicialización

Se editó el script de arranque del sistema (fichero */etc/init.d/rcS*) para que monte el sistema de ficheros de ftrace, *tracefs*, en la ruta standard.

Esto supuso añadir, antes de la línea en la que se ejecuta el shell mediante */bin/ash*, lo siguiente:

```
# NOTA: esto hace accesible el tracefs en /sys/kernel/debug/tracing
/usr/bin/mount -t debugfs nodev /sys/kernel/debug
```

4.3. Instalación de las herramientas

Una vez compiladas, la instalación de las herramientas consistió en copiar los binarios ejecutables (ficheros *tracecmd/trace-cmd* y *stress-ng*) en el directorio */usr/bin* de la distribución.

5. Diseño de las pruebas

En este apartado se resume el diseño de las pruebas realizadas.

Se recogen aquí, por lo tanto, solo las explicaciones principales del funcionamiento y la elección de los comandos finalmente utilizados.

5.1. Construcción de scripts para las pruebas

Durante la construcción de los scripts han resultado de gran utilidad las páginas del *man* de ambas herramientas, especialmente las de *trace-cmd*, pues en ellas hay multitud de ejemplos de mediciones de latencias que han servido de manera directa en la implementación de las pruebas realizadas.

Dadas las características de la placa, se ha tratado de controlar en todo momento la carga de trabajo a la que se la sometía, con el fin de evitar posibles daños debido a un exceso de temperatura. Hay que destacar que las pruebas que aquí se describen se realizaron en el mes de enero, en una habitación refrigerada y durante la noche.

Cada experimento con los scripts entregados se realiza un total de diez veces, pero ese número es configurable cambiando el valor de una variable.

Todos los scripts que se adjuntan siguen la misma estructura, y tan solo cambia entre ellos el tipo de *stressor* y la forma de utilización de la herramienta *trace-cmd*.

5.1.1. Latencia de planificación

En el caso de la latencia de planificación, se han realizado dos mediciones separadas sobre la misma configuración de carga de trabajo: en la prueba cuyo nombre se precede de w"se han usado filtros sobre los eventos relativos a planificación, mientras que en la otra se ha utilizado directamente el tracer *wakeup*. El motivo es que en distintos experimentos se pudo comprobar que el resultado que se obtenía de esas formas era lo suficientemente diferente como para considerar el realizar pruebas por separado.

En lo que respecta al stressor, se ha elegido *–sleep N –sleep-max P"* porque se ha visto que realmente tiene impacto en la latencia de planificación experimentalmente. Su principal ventaja con respecto al resto es que crea threads y no procesos (a diferencia de otros stressors como *fork* o *sem*). Esto permite generar muchas más unidades planificables con los mismos recursos. Dichos threads se despiertan en un tiempo aleatorio, lo cual genera muchos cambios de contexto.

5.1.2. Tiempo máximo de interrupciones inhibidas

En este caso, se ha utilizado directamente el tracer llamado *irqsoff*, que reporta directamente el valor requerido.

En cuanto al stressor, se elige *–sleep N –sleep-max P"* porque empíricamente se ha visto que realmente repercute en el tiempo que las interrupciones están desactivadas. *Sleep* hace lo que *timer*, pero lanzando P threads de sistema por worker, por lo que *timer N* equivale a *sleep N P=1*. No obstante, *sleep* genera mucha más carga con menos coste en memoria, porque son threads, y hacer lo mismo con *timer* supone crear $P * N$ procesos, lo cual supone una sobrecarga excesiva que la placa no es capaz de soportar.

5.1.3. Latencia de llamada al sistema

Para medir la latencia de las llamadas al sistema, se han utilizado filtros sobre los eventos *sys_ enter_ NOMBRE-SC* y *sys_ exit_ NOMBRE-SC*.

Se ha elegido el stressor *tee* porque utiliza tanto *write* como *read*, y lo hace sobre *file descriptors* virtuales, por lo que no supone una sobrecarga innecesaria para la placa debida a la utilización de ficheros físicos.

5.2. Extracción de resultados de las trazas

Una vez con los ficheros de las trazas de la ejecución de cada una de las pruebas, era necesario extraer de ellos los valores de latencia concretos.

Debido a la baja potencia computacional de la placa, durante la ejecución de las pruebas se ha limitado el trabajo a volcar la traza de las mismas en ficheros, que luego serían analizados en una máquina con mayores recursos. El motivo es análogo al que se mencionaba en la sección anterior, ya que a toda costa se ha evitado sobrecargar la beaglebone.

Para llevar a cabo la extracción, se han creado scripts de shell que se encargan de agregar los valores presentes en los ficheros de traza y llevar a cabo ciertos cálculos con ellos.

En primer lugar, está *extract_ latencies.sh*, que sirve para extraer los valores de latencia en aquellas trazas generadas por los tracers *irqsoff* y *wakeup*. Como se puede ver, lleva a cabo un procesamiento sencillo basado en utilidades que funcionan a modo de filtros sucesivos sobre el contenido de las líneas que se analizan.

En el caso de las trazas obtenidas mediante filtros sobre los eventos, se distinguen dos casos:

- En las trazas de las pruebas relativas a las llamadas al sistema, es necesario llevar a cabo un preprocesamiento antes de proceder a la extracción de las latencias. El script *obtain_ latencies.sh* lleva a cabo un parsing de las trazas, emparejando cada evento de tipo *enter* con su correspondiente de tipo *exit*, y calculando entonces el tiempo transcurrido entre sus respectivos timestamps.
- En las trazas de las pruebas denominadas como "scheduling-w", el procesamiento es más sencillo, porque todas ellas terminan con un resumen en el que se da una medición en promedio de la latencia de planificación, por lo que con el script *extract_ latency.sh* dicho valor se extrae mediante utilidades de shell.

Los valores obtenidos se han copiado en hojas de cálculo (que también se adjuntan), y que han servido para elaborar los resultados que se presentan en el siguiente apartado.

6. Resultados

En este apartado se muestran los resultados obtenidos en las pruebas diseñadas en la sección anterior.

Cabe mencionar que todos los valores que se presentan están en microsegundos.

6.1. Tiempo máximo con interrupciones inhibidas

Los resultados agregados en una tabla son los siguientes:

irqsoff								
nº workers	***No Forced Preemption***				***Full Preemption***			
	Avg	**Max**	**Min**	**Std Dev**	**Avg**	**Max**	**Min**	**Std Dev**
4	1024,8	4626	568	1265,70	887	1034	800	67,19
7	820,3	980	758	66,55	1139	1260	999	82,39
10	902,5	1025	857	50,91	1247,9	1339	1170	51,85
13	974,5	1187	830	104,65	1301,8	1552	1209	122,37
16	913,8	1058	800	87,10	1250,8	1633	1082	180,17
19	946	1124	845	90,81	1208,9	1299	1114	60,86
22	977	1050	872	47,65	1209,4	1377	1105	89,49
25	930,6	1009	844	54,34	1300,9	1695	1121	155,78
28	946,7	1024	851	64,50	1192	1331	1124	72,48
31	987,9	1146	845	87,51	1207,3	1315	1033	98,91
34	937,7	1047	856	56,12	1233,9	1485	1102	134,22

Y se corresponden con esta gráfica:

6.2. Latencia de planificación

6.2.1. Latencia de planificación con tracer wakeup

Los resultados agregados en una tabla son los siguientes:

scheduling (tracer wakeup)								
nº workers	***No Forced Preemption***				***Full Preemption***			
	Avg	**Max**	**Min**	**Std Dev**	**Avg**	**Max**	**Min**	**Std Dev**
4	389,3	538	202	132,74	772,7	1296	456	251,54
8	397,8	644	204	145,84	815,7	1147	675	143,76
12	467,7	973	207	202,12	1128,5	1373	956	128,28
16	1070,2	5557	200	1590,72	1359,3	1812	1126	228,47
20	712,2	889	525	114,06	1459,2	1620	1179	118,31
24	710,2	1020	437	189,56	1512,4	1698	1381	91,98
28	759,9	1064	598	145,48	1578,8	1797	1424	130,53
32	911,9	1219	573	211,92	1608,9	1742	1464	85,62
36	746,5	1057	502	180,07	1820,5	2338	1446	288,11
40	953,9	1438	491	287,21	1796,1	2179	1655	157,29
44	1081,6	2437	668	504,39	1791,1	1895	1653	97,70

Y se corresponden con esta gráfica:

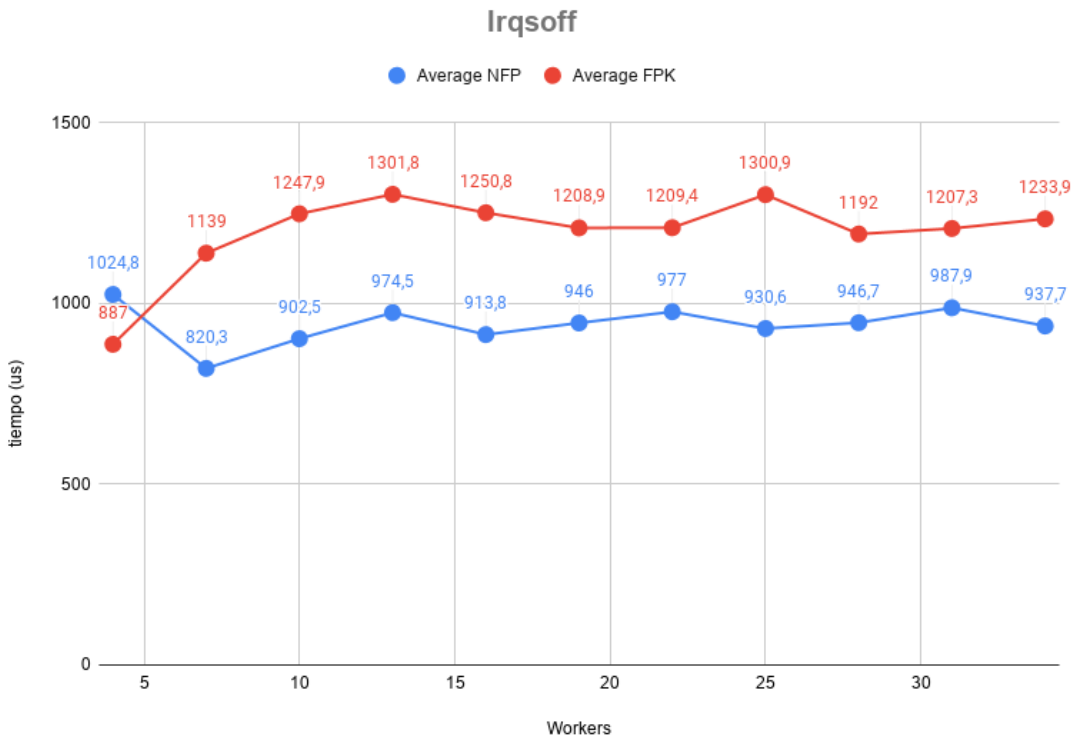

Figura 3: Gráfica comparativa de las medidas de irqsoff

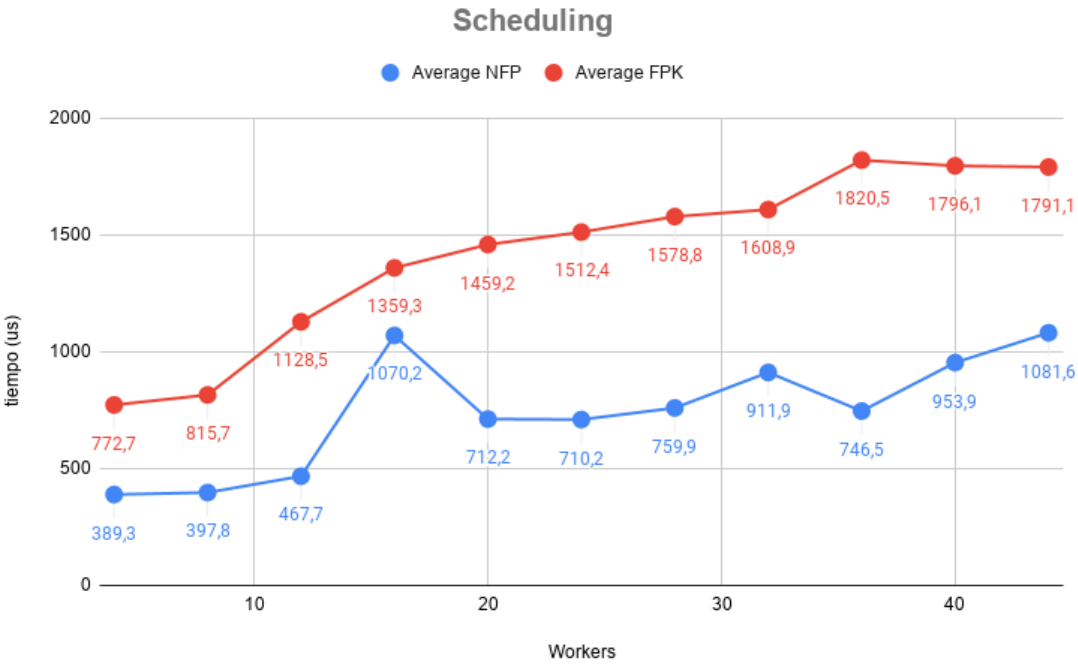

Figura 4: Gráfica comparativa de las medidas de latencia de planificación usando el tracer wakeup

6.2.2. Latencia de planificación con filtros

Los resultados agregados en una tabla son los siguientes:

scheduling (filtros)								
nº workers	*No Forced Preemption*				*Full Preemption*			
	Avg	Max	Min	Std Dev	Avg	Max	Min	Std Dev
4	35.525	42.040	28.744	5234,18	41.053	44.506	33.081	3518,25
8	44.260	53.979	34.509	5227,03	46.738	51.147	42.527	2482,41
12	68.137	92.789	48.091	13179,34	61.326	70.424	53.535	4888,50
16	87.994	99.728	73.820	8181,93	72.344	77.025	69.036	2791,62
20	115.398	133.612	91.740	13953,35	92.517	106.393	78.061	8805,40
24	180.121	206.602	168.037	13399,69	122.896	165.050	101.291	18812,36
28	227.553	250.089	190.498	21691,67	177.471	210.512	149.200	17820,29
32	282.802	302.741	251.664	17189,56	251.706	286.793	229.608	22079,15
36	358.032	369.510	337.391	10478,38	413.173	706.037	300.010	145870,54
40	415.128	430.815	394.227	13862,23	425.685	448.892	407.551	13829,08
44	487.172	507.610	447.544	18537,60	513.896	556.127	492.790	20518,42

Y se corresponden con esta gráfica:

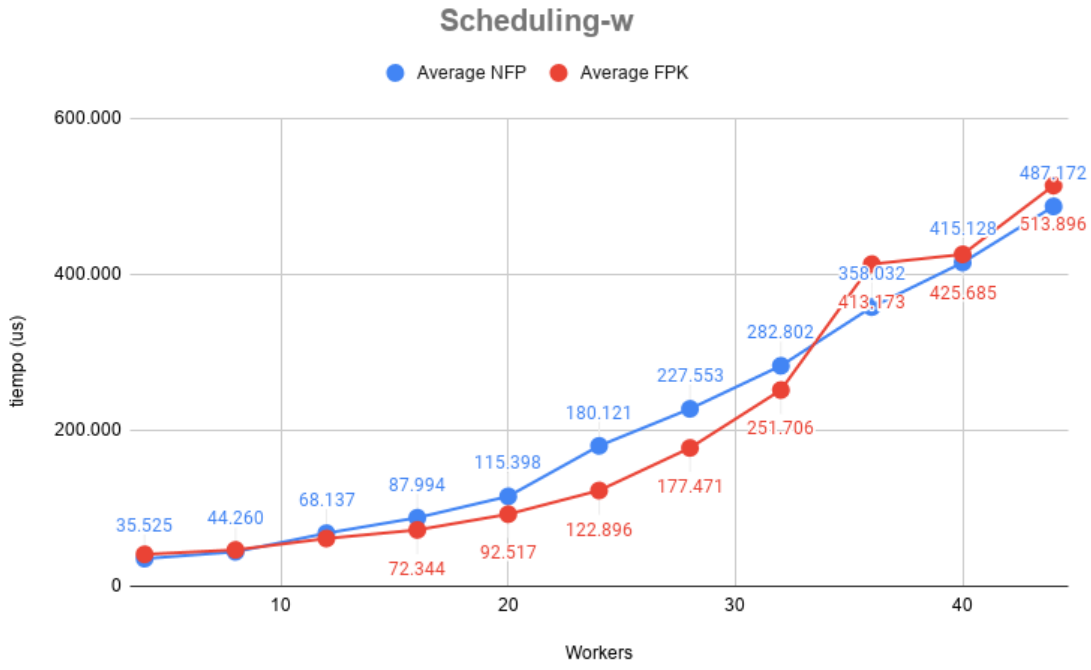

Figura 5: Gráfica comparativa de las medidas de latencia de planificación usando filtros

6.3. Latencia de llamada al sistema

Los resultados agregados en una tabla son los siguientes:

Y se corresponden con esta gráfica:

syscalls								
nº workers	***No Forced Preemption***				***Full Preemption***			
	Avg	**Max**	**Min**	**Std Dev**	**Avg**	**Max**	**Min**	**Std Dev**
1	301,8	419	121	137,80	475	607	450	46,81
3	1172,5	2351	232	839,34	2574,8	2907	2335	158,35
5	3334,2	4005	903	1281,95	4520,2	4753	4442	110,38
7	4602	4614	4575	11,26	6640	6887	5996	293,32
9	5980,4	6007	5926	24,16	8763	8881	8525	122,28
11	7369,6	7410	7344	24,40	10811,4	10918	10587	96,72
13	8730,2	8751	8712	14,10	12717,4	12905	12418	163,14
15	10083,8	10144	10005	46,36	14579,9	14812	14055	267,78
17	11421,9	11483	11325	51,61	16628,1	16744	16458	100,14
19	12800,9	13096	12734	106,50	18512,6	18711	18322	118,34
21	14117,3	14286	13991	103,82	20288,5	20552	19892	202,95

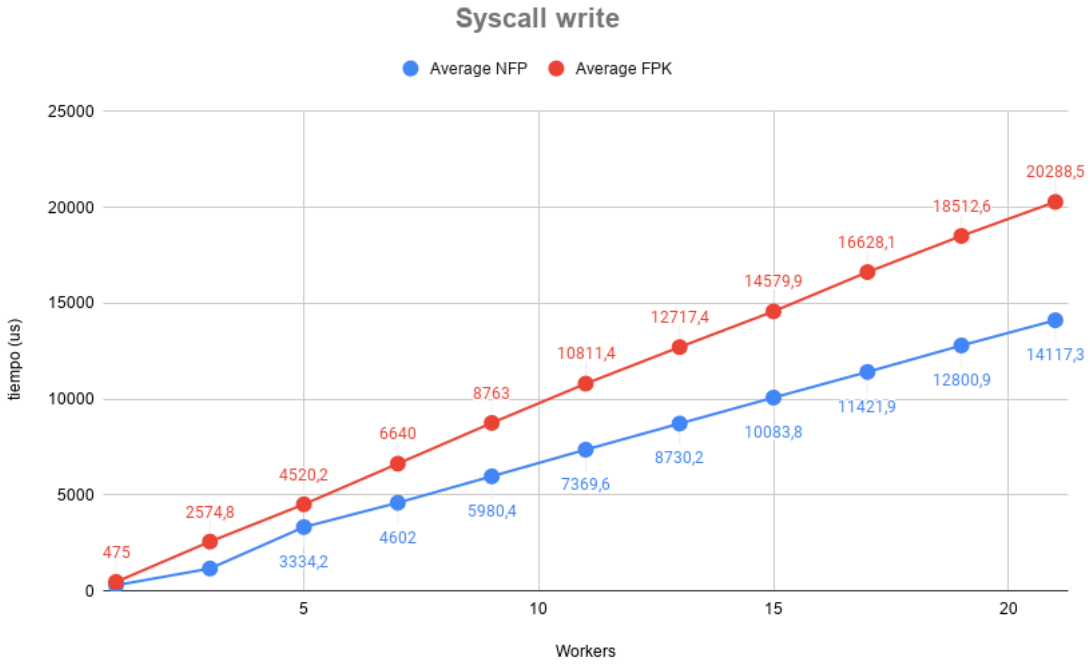

Figura 6: Gráfica comparativa de las medidas de latencia de llamada al sistema

7. Conclusiones

En este apartado se van a analizar los resultados obtenidos en los experimentos y se va a llevar a cabo una reflexión sobre las posibles mejoras en el trabajo realizado.

7.1. No Forced Preemtion vs. Fully Preemptible Kernel

En las tablas y las gráficas se puede observar como el kernel configurado con *full preemption* presenta latencias más altas que el configurado con *no forced preemption*, lo cual es coherente con el hecho de que el planificador esté siendo invocado con una mayor frecuencia en el primero de ellos. Además, el contraste es todavía más evidente si se tiene en cuenta que con esquema de expulsión *no forced preemption* es con el que menos intervenciones del scheduler se realizan.

En ambos casos existe una gran variabilidad entre las latencias medidas en las diferentes experimentaciones realizadas, siendo este hecho evidente al analizar la desviación típica de cada una de las pruebas. Si bien puede considerarse que algunas de las ejecuciones den medidas espurias, el hecho de que ocurra para casi todas las pruebas hace que se deba considerar más bien como una tónica general. A diferencia de lo que cabría esperar, la configuración *full preemption* presenta una variabilidad similar a la *no forced preemption*, incluso cuando la carga de trabajo es elevada.

Por lo tanto, a la vista de los resultados, no parece que el uso de Linux RT en esta plataforma en particular sea una buena idea si lo que se pretende es obtener garantías tiempo real.

7.2. Posibles mejoras

El tiempo de ejecución del script *obtain_latencies.sh* es especialmente elevado, en parte por el tamaño de las trazas que se analizan, pero también debido a que el procesamiento se está llevando a cabo con bash. Sería interesante migrar el script a Python o a otro lenguaje con mejores prestaciones, ya que si bien Python seguiría siendo lento, mejoraría considerablemente el tiempo de ejecución con respecto a bash.

Por otra parte, me habría gustado analizar las trazas obtenidas usando *kernel shark*, pues he visto que sirve como visor de los ficheros *trace.dat* que se generan con *trace-cmd*, y debe ser especialmente útil combinado con el tracer *function_graph*.

Referencias

[1] *Blog sobre el funcionamiento de ftrace en Linux.* URL: `https://embeddedbits.org/tracing-the-linux-kernel-with-ftrace/` (visitado 05-02-2021).

[2] *Blog sobre kprobes en Linux.* URL: `https://vjordan.info/log/fpga/how-linux-kprobes-works.html` (visitado 05-02-2021).

[3] *Blog sobre tecnologías de tracing en Linux.* URL: `https://jvns.ca/blog/2017/07/05/linux-tracing-systems/` (visitado 05-02-2021).

[4] *Documentación oficial de Ftrace en la versión más reciente de Linux.* URL: `https://www.kernel.org/doc/html/latest/trace/ftrace.html` (visitado 05-02-2021).

[5] *Documentación oficial de la herramienta stress-ng.* URL: `https://kernel.ubuntu.com/~cking/stress-ng/stress-ng.pdf` (visitado 05-02-2021).

[6] *Documentación oficial de tecnologías de tracing en la versión más reciente de Linux.* URL: `https://www.kernel.org/doc/html/latest/trace/index.html` (visitado 05-02-2021).

[7] *Documentación oficial del kernel 3.9 sobre Ftrace. (relacionado: https://elixir.bootlin.com/linux/v3.9/source/Documentation/trace/events.txt y https://elixir.bootlin.com/linux/v3.9/source/Documentation/trace/tracepoints.txt).* URL: `https://elixir.bootlin.com/linux/v3.9/source/Documentation/trace/ftrace.txt` (visitado 05-02-2021).

[8] *Fuentes de Linux con el parche tiempo real aplicado.* URL: `sftp://hendrix.cps.unizar.es/misc/practicas/SE-II/PR3/kernel-rt.xz` (visitado 31-01-2021).

[9] *Páginas de manual de la herramienta trace-cmd.* URL: `https://kernel.ubuntu.com/~cking/stress-ng/stress-ng.pdf` (visitado 05-02-2021).

[10] *README de la herramienta stress-ng para plataformas Android.* URL: `https://github.com/ColinIanKing/stress-ng/blob/master/README.Android` (visitado 05-02-2021).

[11] *Repositorio oficial de la herramienta stress-ng.* URL: `https://github.com/ColinIanKing/stress-ng` (visitado 31-01-2021).

[12] *Repositorio oficial de la herramienta trace-cmd.* URL: `git://git.kernel.org/pub/scm/utils/trace-cmd/trace-cmd.git` (visitado 31-01-2021).