Universidad Zaragoza

1542

Trabajo Fin de Máster

Caracterización del comportamiento y gestión de interrupciones en sistemas empotrados Linux sobre arquitecturas ARM

Memoria Principal

Autor/es

Iván Rodríguez Perales

Director/es

José Luis Briz Velasco

Escuela de Ingeniería y Arquitectura

2013-2014

Resumen

El objetivo de este TFM es caracterizar el comportamiento de las interrupciones sobre SoCs con núcleo ARM orientado a aplicaciones, gestionados mediante Linux. Para ello se parte del estudio de todos los posibles mecanismos hardware disponibles en ARM para gestionar y disminuir la latencia de interrupción. Se estudia el tratamiento de las excepciones en el núcleo de Linux, prestando especial atención a la influencia de los diferentes modos de expulsión, incluyendo la expulsión total para tiempo real estricto, en condiciones de carga y sin carga. Sobre dos placas (Beaglebone y Raspberry Pi) con diferentes microprocesadores ARM, en ambos casos orientados a aplicación, se caracterizan las latencias en diferentes niveles y condiciones, desde la generación /retorno de la señal hasta los handlers y rutinas de servicio de niveles más altos del núcleo. Se estudian las diferencias en los tiempos de respuesta de las dos placas analizadas en diferentes condiciones. Se muestra que la programación *baremetal* sin sistema operativo, recurriendo a librerías optimizadas por el fabricante, puede ser más estable pero poco eficiente, proporcionando una latencia de respuesta en torno a 1.5 ms, en relación a Linux/ARM que proporciona latencias de respuesta medias en torno a 10 us. Se estudian las variaciones en la latencia de interrupción que aparecen en Linux/ARM y se concluye por exclusión que puede deberse a la gestión de los gpios y el sistema de entrada/salida de las placas, y no con los controladores de interrupciones del SoC o con los mecanismos de gestión de Linux.

Índice

1. Introducción

Los procesadores ARM cubren todo el rango demandado por las aplicaciones empotradas, desde el segmento de microcontroladores sencillos, hasta el de microprocesadores superescalares que soportan memoria virtual, generalmente incluidos en sistemas on-chip (SoCs) de alta funcionalidad y bajo consumo. La utilización de núcleos Linux sobre arquitecturas ARM ha venido incrementándose de forma espectacular, compitiendo incluso en el terreno de los sistemas Tiempo Real. A ello está contribuyendo la multiplicación de placas de bajo coste dotadas de SoCs con núcleos ARM, en las que Linux es bien el sistema operativo por defecto, bien una alternativa fácil de portar. Estas placas están ocupando rápidamente el segmento de los micro-PCs industriales o los wall-plug systems.

Linux, sin embargo, no deja de ser un sistema de propósito general lejos de la simplicidad de sistemas tiempo real específicos como VxWorks, uCOS, SYS/BIOS etc, que permiten un gran control sobre parámetros críticos como las latencias de interrupción y planificación. Esta generalidad de Linux le convierte en inevitablemnte complejo: el núcleo monolítico puede configurarse con modos de expulsión muy diferentes, y para arquitecturas mono y multiprocesador /multinúcleo. La complejidad se multiplica por el amplio rango de opciones de la familia ARM en sus tres segmentos (microcontroladores, tiempo real y aplicaciones). Ello hace que, en la práctica, el desarrollo sobre plataformas Linux/ARM, aparentemente asequibles, requiera esfuerzos superiores a lo inicialmente previsto.

El objetivo de este TFM es caracterizar el comportamiento de las interrupciones sobre SoCs con un núcleo ARM orientados a aplicaciones, gestionados mediante Linux, considerando sus diferentes modos de expulsión. Se pretende caracterizar adecuadamente esta latencia desde diferentes niveles (desde la generación /devolución de señal hasta handlers / rutinas de servicio de niveles más altos del núcleo), teniendo en cuenta distintos modos de expulsión, a fin de estudiar las posibilidades de optimización que puedan extraerse de la arquitectura de sistema ARM.

La memoria se estructura en la siguiente secciones. La Sec. 2 explica las características de la gestión de interrupciones en ARM y Linux, y los modelos de expulsión del núcleo de Linux. Las Secs. 3 y 4 describen respectivamente el entorno de trabajo utilizado y la metodología aplicada. La Sec. 5 expone y discute los resultados, y la Sec. 6 cierra con las correspondientes conclusiones.

2. Fundamentos

2.1. Factores que influyen en la latencia de interrupción en ARM

En el contexto que nos ocupa, una interrupción es una excepción asíncrona en relación a la ejecución de las instrucciones de un programa, producida por un dispositivo externo y atendida por el kernel. La latencia referida en este trabajo será la latencia de interrupción, esto es el tiempo que transcurre desde que una señal es recibida en el sistema hasta que llega al proceso que se encarga de gestionarla. La lista de factores que afectan a la latencia dependerá de las características de la plataforma hardware (procesador y periféricos) y de cómo gestiona estos eventos el sistema operativo utilizado.

2.1.1. Mecanismo básico vectorizado. FIQ y IRQ.

La arquitectura ARM asocia un código a cada excepción a través de una tabla, almacenada en memoria principal, conocida como tabla de vectores de excepción. La tabla para el caso de Linux puede verse en el código 1. Cada entrada contiene típicamente una instrucción de carga del contador de programa (pc), bien una instrucción load bien un salto, según la dirección de inicio del vector de interrupciones. Las dos últimas entradas corresponden a las dos interrupciones soportadas por cualquier sistema ARM, IRQ y FIQ, respectivamente asociadas a sendas entradas diferenciadas del procesador. La FIQ (Fast Interrupt Request) tiene siempre mayor prioridad que la IRQ (Interrupt Request), y al corresponder al último elemento de la tabla del vector de interrupciones es posible almacenar hasta 4 KB de instrucciones, en lugar de una única instrucción de carga del pc, que se ejecutarán cuando ocurra la FIQ. En muchos SoCs la FIQ se reserva para usos internos del sistema y no puede programarse.

Este mecanismo presente en cualquier ARM supone los siguientes pasos:

- Almacenamiento automático de la palabra de estado, del pc y de los registros r13 (sp) y r14 (lr), con cambio a modo protegido, que incluye la inhibición de la IRQ, y también de la FIQ si se está tratando esta última.
- Búsqueda y ejecución por parte del procesador de la instrucción de modificación del pc asociada a la IRQ/FIQ en el vector de interrupciones
- Ejecución del código al que se transfiere el control en el paso anterior

```
        .globl  __vectors_start                                          1
__vectors_start:
 ARM(   swi     SYS_ERROR0      )                                        3
 THUMB( svc     #0              )
 THUMB( nop                     )                                        5
   W(b) vector_und + stubs_offset   @ Undefined ins.
   W(ldr) pc, .LCvswi + stubs_offset  @ SWI                              7
   W(b) vector_pabt + stubs_offset  @ Prefetch abort
   W(b) vector_dabt + stubs_offset  @ Data abort                         9
   W(b) vector_addrexcptn + stubs_offset
   W(b) vector_irq + stubs_offset  @ IRQ                                 11
   W(b) vector_fiq + stubs_offset  @ FIQ
```

Código 1: Tabla del vector de excepciones

Los dos primeros pasos están optimizados en implementaciones de ARM típicas para que se realicen en 4 ó 6 ciclos. Pero en presencia de varios dispositivos con capacidad de enviar interrupciones, éstos han de compartir la única IRQ existente. Por tanto el código asociado a la única IRQ ha de encuestar los dispositivos que comparten la línea para averiguar cuál de ellos ha interrumpido, y ejecutar un código específico en cada caso. Denominaremos en lo que sigue *handler* al código inmediatamente asociado a la IRQ/FIQ, y *rutina de servicio a interrupción* (ISR) a la rutina que trata la interrupción de un evento específico. Este procedimiento tiene un coste alto, y por ello los diferentes segmentos de microprocesadores ARM incluyen optimizaciones diversas, especialmente controladores de interrupciones, de los que se hablará más adelante.

En términos de latencia, la FIQ está optimizada para evitar una transferencia de control, está dotada de mayor prioridad por hardware, y por tanto responde siempre con una latencia menor. La Fig. 1 expresa gráficamente la diferencia entre la respuesta a una FIQ y una IRQ.

2.1.2. Controladores de interrupciones

La mayoría de sistemas ARM incorporan controladores de interrupciones vectorizados que facilitan el reconocimiento de la interrupción, la localización de la ISR y la gestión de prioridades entre interrupciones(ver capítulo 7.2 de los Anexos). Los GIC tienen opciones más avanzadas que los VIC y soportan procesadores multicore.

2.1.3. Mecanismos hardware relacionados

Como también se puede apreciar en la figura 1, la latencia estará condicionada por las características físicas de los periféricos y procesador. Solamente

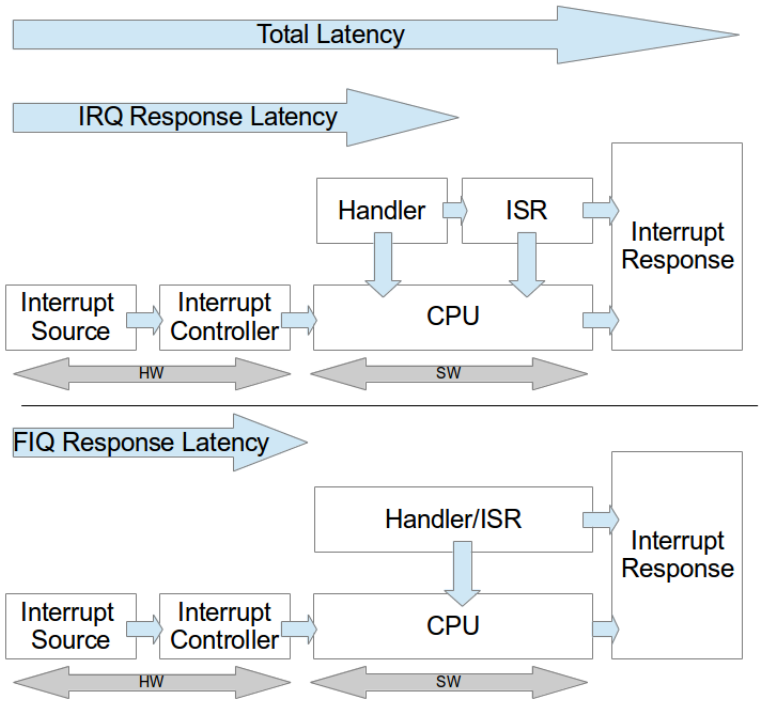

Figura 1: Tiempos de respuesta ante una IRQ y FIQ

se podrá influir sobre el hardware si estos dispositivos ofrecen mecanismos para su configuración. En ARM existen mecanismos configurables que describimos brevemente.

- Predicción de saltos. Los microprocesadores ARM orientados a aplicaciones, como los Cortex A, disponen de mecanismos de predicción de saltos, que suponen la ejecución especulativa de código. En general este mecanismo incrementa notablemente el rendimiento, pero cuando la predicción es incorrecta hay que expulsar todas las instrucciones de la rama equivocada, con un descenso del importante en el rendimiento en aplicaciones en las que la predicción tiende a fallar. En sistemas tiempo real la predicción de saltos puede llevar a diferencias importantes en la cota de peor tiempo de ejecución (WCET) calculado. Los microprocesadores ARM con esta capacidad permiten también desconectarla.

- Jerarquía de Memoria. Muchos microprocesadores ARM incorporan memorias cache, que introducen una gran variabilidad en los tiempos de ejecución de tareas y rutinas de servicio a excepción, en función de la tasa de fallos en cache y de la lejanía en ciclos de la memoria principal (del orden de cientos de ciclos en general). Estas caches pueden ser

no bloqueantes (*lock-up free caches*), pudiendo servir nuevas peticiones que aciertan mientras se atienden fallos pendientes (*Hit Under Miss* (HUM)). Los fallos pendientes corresponden a instrucciones anteriores en orden de programa a las que siguen siendo servidas mientras aciertan. Ante una excepción, es preciso poder recuperar el *estado preciso*, reanudando la ejecución a partir de la instrucción coherente con el orden de programa. Este mecanismo tiene un cierto coste. En ARM es posible desactivar tanto las caches presentes como el mecanismo HUM.

- Memoria virtual. El soporte a la gestión de memoria virtual en núcleos ARM orientados a aplicaciones está compuesto por la Memory Management Unit (MMU) y los Translation Lookaside Buffers (TLBs). Linux mantiene código y datos del núcleo en memoria física, y aunque es posible desde el código del núcleo acceder a páginas de procesos de usuario, esto no está permitido ni en las rutinas de servicio a excepción ni las funciones diferidas activadas desde ellas. Sin embargo, dado que todas las direcciones lanzadas por el procesador pasan por el mecanismo de traducción, los fallos en TLB con intervención de la MMU pueden suponer incrementos en las latencias de ejecución de esas rutinas (ver Sec. 7.1 del Anexo).

- *Low latency bit* o *FI bit*. Al activar este bit en una plataforma que lo soporte, el procesador realiza en unos pocos ciclos menos el cambio de estado y el salto a la instrucción asociada en el vector de interrupción. El ligero ahorro en ciclos de este mecanismo proviene de anular caches o los mecanismos especulativos como la predicción de saltos o el HUM. Los mecanismos particulares afectados por el bit dependen de cada microprocesador, y en general no están documentados. La anulación de los mecanismos especulativos permite evitar la gestión de excepciones precisa, pero tiene como efecto lateral impedir el uso de operaciones de load/store múltiple que no son completamente reiniciables. Esto supone compilar el sistema operativo de forma que no se generen loads y stores múltiples. Hemos intentado hacerlo en este trabajo sin éxito, y no está claro que pueda llevarse a cabo al menos con la *toolchain* utilizada. En definitiva, Aunque provoca una disminución de la latencia, el rendimiento del sistema descenderá, sobre todo si se desactivan las caches.

2.2. Características del núcleo de Linux

Linux es un derivado de Unix y su kernel tiene un diseño monolítico: consta de un único binario, aunque con la posibilidad de enlazado dinámico

de módulos, que se ejecuta con el procesador en modo protegido, capaz de localizar, cargar, ejecutar y controlar otros binarios (procesos) que se ejecutan con el procesador en modo usuario. El núcleo proporciona un interfaz de gestión de abstracciones tales como ficheros y procesos, y de recursos hardware como la memoria y los dispositivos de e/s en general. La ejecución de los procesos provoca excepciones en el procesador. Pueden hacerlo de forma directa, a veces intencionada y previsible (para solicitar un servicio del núcleo), a veces no intencionada ni previsible (fallos de página, operaciones ilegales). También puede hacerlo de forma indirecta, como es el caso en interrupciones asíncronas derivadas de operaciones de e/s. Todas estas excepciones suponen un cambio de estado (a modo protegido) y la ejecución de código del núcleo, que comienza y acaba —antes de regresar a modo usuario para reanudar el código de un proceso— en una rutina de servicio a excepción. Esta ejecución de código de núcleo o *kernel activiy* y que consta de una cascada de invocaciones a funciones internas del mismo, se suele denominar *Kernel Control Path* (KCP). En un núcleo con expulsión parcial, los KCPs se comportan como corrutinas, entrelazando su ejecución sea cediéndose el control unos a otros, sea mediante mecanismos de interrupción. En un núcleo con expulsión total los KCPs no se entrelazan sino que se planifican como tareas independientes, incluidos los iniciados como respuesta a interrupciones, sujetos a prioridades junto al resto de procesos del sistema.

2.2.1. Gestión de interrupciones en Linux

La gestión software de una IRQ parte de la rutina de la tabla del vector de interrupciones (ver código 1) y sigue un *fall through* que variará levemente en las primeras funciones dependiendo de las características del hardware (Código 2):

```
@@ Secuencia rutinas en ASM
vector_\name
__irq_svc
irq_handler
arch_irq_handler_default

// Secuencia de rutinas en C
asm_do_IRQ
handle_IRQ
generic_handle_irq
generic_handle_irq_desc
handle_irq_event
handle_irq_event_percpu
ISR
```

Código 2: Secuencia de llamadas del handler de una IRQ

Por el contrario, una FIQ sólo requiere una rutina en ensamblador (ver Código 3) que deberemos asociar a la entrada correspondiente de la tabla del vector de interrupciones.

```
.text

ENTRY(test_fiq_handler)
   ...
test_fiq_handler_end:
END(test_fiq_handler)
```

Código 3: Rutina asociada a la entrada del vector de interrupciones de la FIQ

En sistemas multitarea como Linux, las interrupciones nunca pueden bloquearse, es decir, no pueden invocar al planificador y provocar un cambio de contexto. Esto significa que durante su ejecución monopolizan el uso del procesador, por lo que la latencia tiene que ser la menor posible. Los sistemas operativos cuentan con varios mecanismos (funciones diferidas o *callouts*) para evitar estos bloqueos. La Sec. 7.1 del Anexo resume brevemente las existentes en Linux

Por este motivo, el retorno de una excepción es un punto crítico en el núcleo de Linux. La ISR retorna al handler mediante un retorno convencional de subrutina, pero el handler salta a una secuencia de retorno común para todas las interrupciones en la que se comprueba si hay interrupciones anidadas. Si no las hay, se restaura el contexto y se reanuda la ejecución interrumpida, mediante la ejecución de una instrucción que restaura el estado que se almacenó al producirse esa interrupción. Si por el contrario hay interrupciones anidadas, se pueden ejecutar las funciones diferidas pendientes, e incluso invocar al planificador provocando un cambio de contexto.

Conviene puntualizar que recientemente los desarrolladores del kernel[1] han apostado por no permitir el anidamiento de interrupciones. Por tanto, durante la ejecución de un proceso de atención a una IRQ o FIQ, no se producirán tampoco cambios de contexto de bajo nivel que puedan provocar un aumento de latencia. La única excepción es una FIQ, que puede expulsar a una IRQ en ejecución.

2.2.2. Modos de expulsión del núcleo y gestión de excepciones

El modelo de expulsión del kernel determina cómo se va a ejecutar un proceso del sistema incluyendo excepciones y por tanto, interrupciones. Existen varios modelos:

[1] `http://lwn.net/Articles/380931/`

- *Expulsión parcial (No forced preeemption - PREEMPT_ NONE):* Una interrupción no puede ser expulsada cuando se ejecuta en su *top half* a no ser que explícitamente invoque al planificador(ver capítulo 7.1 del Anexo).
- *Expulsión voluntaria (Voluntary Kernel Preemption - PREEMPT_ VOLUNTARY):* Existen varios puntos en la ejecución del proceso que le permiten invocar al planificador y ser expulsado
- *Núcleo expulsivo (Preemptible kernel - PREEMPT_ DESKTOP)*: Un proceso puede expulsar a otro de menor prioridad que se encuentre en ejecución cuando accede aun *spin-lock*.
- *Tiempo real (Complete preemption - PREEMPT_ RT):* Cualquier proceso en ejecución puede ser expulsado por el planificador. Las interrupciones son tratadas como *threads del kernel* y son planificadas como cualquier otro proceso

Cuando una interrupción retorna de ejecución pasa a través de un mecanismo que, o bien (1) activa una excepción que estuviese anidada, (2) ejecuta *bottom halfs* pendientes o (3) invoca al planificador.

3. Entorno experimental

Se han utilizado dos placas de desarrollo que montan un SoCs con un núcleo ARM para realizar el estudio, una Raspberry (ver tabla 1) y una BeagleBone (ver tabla 2). Ambas soportadas por distintas versiones del kernel de Linux. No ha sido posible realizar las pruebas utilizando exactamente la misma versión del kernel, debido en parte al *roadmap* de desarrollo de los kernels para esos sistemas y a su grado de estabilidad.

Placa Raspberry[2]	
Familia:	ARM11
Arquitectura:	ARMv6KZ
Núcleo:	ARM1176JZ(F)-S [3]
SOC:	BCM2835 [4]
Versión del kernel linux:	3.12.24+
Versión del kernel linux RT:	3.12.24+ con parche PREEMPT_RT[5]

Tabla 1: Especificaciones de la placa Raspberry y distribución Linux utilizada

Placa BeagleBone[6]	
Familia:	Cortex-A
Arquitectura:	ARMv7-A
Núcleo:	Cortex-A8 [7]
SOC:	AM335x [8]
Versión del kernel linux:	3.14.23+
Versión del kernel linux RT:	3.14.23+ con parche PREEMPT_RT [9]

Tabla 2: Especificaciones de la placa BeagleBone y distribución Linux utilizada

Para las mediciones de señal se ha hecho uso de un osciloscopio Tektronix TDS 2002B [10].

Entorno de desarrollo	
Sistema Operativo:	Ubuntu 12.04.5
Editor de código:	vim
Toolchain Raspberry:	arm-linux-eabi
Toolchain Beaglebone:	arm-linux-eabihf
Crosscompiler:	gcc version 4.71
IDE Sys/BIOS(Prueba Bare-Metal):	CCS 5.5 + StarterWare

Tabla 3: Listado de herramientas utilizadas en el proyecto

4. Metodología

4.1. Toma de muestras

Para poder caracterizar la latencia de la interrupción tenemos que utilizar un sistema preciso de medida de tiempo. Además, deberá realizar mediciones en diferentes puntos del código del *handler* e ISR de la interrupción.

Existen dos métodos básicos:

[2] `http://www.raspberrypi.org/`
[3] `http://www.arm.com/products/processors/classic/arm11/arm1176.php`
[4] `http://www.broadcom.com/products/BCM2835`
[5] `https://rt.wiki.kernel.org/index.php/CONFIG_PREEMPT_RT_Patch`
[6] `http://beagleboard.org/bone`
[7] `http://www.arm.com/products/processors/cortex-a/cortex-a8.php`
[8] `http://www.ti.com/product/am3358`
[9] `https://rt.wiki.kernel.org/index.php/CONFIG_PREEMPT_RT_Patch`
[10] `http://www2.tek.com/cmswpt/psdetails.lotr?cs=psu&ci=13295&lc=ES`

- *Externo, no invasivo, o de caja negra*, utilizando un osciloscopio o un analizador lógico, que registre el cambio de nivel de tensión producido en un pin del microprocesador. Por ejemplo puede registrarse la entrada de una señal por un gpio que se ha asociado a una IRQ, y la salida por el mismo y otro gpio de una señal generada desde la parte del núcleo hasta la que queremos realizar la medición. Existe una sobrecarga debida a la escritura en el gpio de salida, muy difícil de caracterizar.
- *Interno, invasivo o de caja blanca*, mediante lecturas de un temporizador alta resolución, una al comienzo y otra al final de la parte del núcleo que queremos temporizar. A la hora de medir latencia de respuesta es menos precisa, ya no podemos tomar una medida hasta después de ejecutar un número significativo de instrucciones del handler asociado, ya que como mínimo hay que guardar parte del estado no salvado por el hardware pasando a modo svc (Sec. 2 Código 2. Sin embargo tiene la ventaja de que puede registrarse un número arbitrario de medidas durante el tratamiento de una interrupción.

Los puntos de mayor interés serán las entradas y salidas de las secuencia de funciones que atraviesa una IRQ y una FIQ(ver apartado 2.2.1) ya que nos permiten modelar la latencia con respecto tanto al momento en que se lanzó el evento que generaba la interrupción como a cualquier otro punto de interés de la secuencia.

4.1.1. Registro de tiempo mediante osciloscopio

Para realizar esta medida generamos una onda cuadrada. El flanco de subida se envía a través de un pin de la placa en el momento en el que se inicia la prueba, a cuyo gpio se ha asociado la IRQ y la correspondiente ISR. El flanco de bajada se envía desde el punto del núcleo de Linux del que deseamos medir la latencia de respuesta (handler, ISR, función diferida o incluso código de usuario en función del sistema de que se trate).

4.1.2. Registro de tiempo de alta resolución

Los dos núcleos ARM poseen un registro llamado *Cycle Counter Register* en el coprocesador CP15 de control, que cuenta el número de ciclos de reloj del microprocesador. El modelo que utilizamos utiliza el reloj de frecuencia de cada procesador [11]:

$$tiempoCiclo_{raspberry} = 1/700MHz = 1'428ns \quad (1)$$

[11] $ cat /sys/devices/system/cpu/cpu0/cpufreq/scaling_cur_freq

$$tiempoCiclo_{beaglebone} = 1/720MHz = 1'388ns \quad (2)$$

Para poder leer el registro de alta precisión con sobrecarga mínima utilizamos la directiva *volatile*, que evita que el compilador realice optimizaciones mediante planificación estática sobre esa instrucción o secuencia asm:

```
static inline unsigned ccnt_read (void)
{
  unsigned cc;
  asm volatile ("mrc␣p15,␣0,␣%0,␣c15,␣c12,␣1" : "=r" (cc));
  return cc;
}
```

4.1.3. Datos y métricas

A partir de las lecturas obtenidas en varios de los puntos del tratamiento de la IRQ o de la FIQ podemos averiguar las etapas que presentan una mayor latencia o variabilidad, que pueden ser más determinantes en el diseño de un sistema con restricciones de tiempo real.

Para que las mediciones sean robustas, se lanzan n interrupciones y se obtienen n lecturas en cada uno de los tests y de las etapas. A partir de los datos obtenidos se calculan las siguientes métricas:

- Media (Eq. 3): Evidencia las etapas han generado una mayor latencia.

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} a_i = \frac{a_1 + a_2 + \cdots + a_n}{n} \quad (3)$$

- Desviación típica (Eq. 4): Nos permite ver el grado de dispersión de las muestras obtenidas en cada etapa. Gracias a esta información podremos saber que etapas son más estables en términos temporales.

$$s = \sqrt{\frac{1}{N}\sum_{i=1}^{n}(x_i - \bar{x})^2} \quad (4)$$

- Desviación típica relativa(ver 5): Relaciona la desviación típica con la media y nos da un porcentaje que indica la fiabilidad del cálculo estadístico. Cuanto menor sea el porcentaje, menor desviación existirá.

$$\%RSD = \frac{s}{\bar{x}} \times 100 \quad (5)$$

Esta información la ofreceremos desde dos punto de vista, (1) la latencia desde el momento en que se lanza la interrupción hasta que se llega al punto donde tomamos la medida; y (2) la latencia entre puntos de registro de tiempo.

Por último, existe información adicional que podemos extraer de los registros del procesador ARM que nos ofrecerán información muy valiosa sobre algunos de los elementos que influyen en la latencia y que también obtenemos por cada una de las pruebas realizadas a través del coprocesador CP15 de ARM[12]:

- Fallos de TLB
- Fallos en la cache de instrucciones
- Fallos en la cache de datos
- Fallos en la predicción de saltos

4.1.4. Experimentación con carga y sin carga

Para estudiar la diferente respuesta de los distintos métodos de expulsión del kernel en sistemas con carga alta de procesos frente a sistemas sin carga, se han definido y creado las condiciones respectivas de la forma siguiente:

- *Sistema sin carga*: Se anula la ejecución en arranque de todos los daemons a nivel de usuario, de modo que las rutinas de servicio a interrupción sólo compiten por el procesador con los threads de kernel (*kswapd, ksoftirqdn* etc). Estos threads permanecen en general bloqueados y se ejecutan en su caso bien con muy baja prioridad, bien únicamente en circustancias críticas del sistema (muy baja memoria física disponible en el caso de *kwapd* por ejemplo).
- *Sistema con carga*: Se lanzarán cuatro procesos adicionales que estarán ejecutando en todo momento operaciones que hagan un uso intensivo del procesador, como *sqrt*, y por tanto estén compitiendo directamente por ese recurso con cualquier otro proceso que se esté ejecutando en ese instante.

[12]Aunque se le denomina de este modo, CP15 no es un coprocesador como tal sino un interfaz de acceso a registros de monitorización incluidos en los microprocesadores ARM, cuyas funcionalidades varían según el modelo

Pruebas realizadas	
Normal	Medida en un sistema con carga baja
Carga	Prueba en un sistema con varios procesos copando el procesador [13]
No predicción de saltos:	Rendimiento de la latencia en un procesador con la predicción de saltos deshabilitada
No caches	Tiempo de latencia de interrupción en un sistema con las caches deshabilitadas

Tabla 4: Pruebas realizadas para cada uno de los cuatro modos de expulsión en cada placa

4.1.5. Programación *bare-metal*

Es difícil si no imposible sincronizar medidas de osciloscopio con las lecturas de tiempo obtenidas de los temporizadores del microprocesador. El tramo temporal definido entre la entrada de la señal a la placa y la primera lectura de tiempo escapa a toda posibilidad de medida. Una manera indirecta de aislar este tramo es la programación bare-metal de la placa Beaglebone, sin sistema operativo. Para ello se han utilizado las librerías del paquete Starterware de Texas Instruments, por considerarse que están especialmente optimizadas para los productos de la compañía. La programación y generación del ejecutable de ha realizado mediante Code Composer Studio (CCS) 5.5. El binario se ha generado en modo *release* a fin de obtener un código optimizado. La carga se ha efectuado desde CCS a través del JTAG presente en Beaglebone.

5. Resultados

La tabla 4 resume el conjunto de pruebas realizadas de caracterización de la latencia de interrupción. Estas pruebas se han realizado en cada uno de los modelos de expulsión del kernel y en cada una de las dos placas. Las medidas se han tomado en los dos sistemas de medición, explicados en el apartado 4, para verificar que eran correctas. Por comodidad, las muestras aquí mostradas son las adquiridas a través del registro de tiempo de alta resolución.

La tabla 5 resume las pruebas que no ha sido posible realizar y las razones correspondientes. Salvo las excepciones indicadas en la tabla, se trata de pruebas no han podido realizarse ni en BeagleBone ni en Raspberry.

Pruebas que no ha sido posible llevar a cabo	
No MMU	Aunque desde el menú de configuración del kernel es posible deshabilitar el uso de la memoria virtual, varios errores durante la compilación del kernel reflejan dependencias a funciones del sistema de memoria virtual que no permiten generar un kernel válido
Low latency interrupt (RB)	Tal y cómo se describe en el apartado 2.1.3 se debe generar código que no haga uso de operaciones de load y store múltiples. Esto se consigue con un flag del compilador gcc que no fue reconocido por el crosscompiler de nuestro entorno de trabajo(ver capítulo 3)
FIQ	Cuando se describe este tipo de interrupción en el apartado 2.1.1, se menciona que las FIQ se utilizada por funciones básicas del sistema y el usuario no puede hacer uso de ellas (BB). En la RB, ésta está dedicada a la gestión del host USB y puede liberarse. El problema aparece cada vez que es invocada ya que bloquea todo el sistema
Caches (BB)	Deshabilitar las caches provoca que la BB sea incapaz de ejecutarse, lanzado un kernel panic

Tabla 5: Listado de pruebas no realizadas en el trabajo. BB: Beaglebone; RB: Raspberry Pi

5.1. Comparativa de latencias de interrupción

En la Fig. 2, y con más detalle en la tabla 6, se muestra el modo de expulsión por plataforma que ha respondido antes en media a una interrupción bajo unas condiciones determinadas (indicadas por A, B y C). La barra CCS/BB es el tiempo de latencia de respuesta de una ISR, obtenido mediante programación de la placa BeagleBone *bare-metal* (Sec. 4.1.5). Puede observarse que los modos de núcleo expulsivo y voluntaria presentan las menores latencias.

La Fig. 3 y recoge aquellos modos de expulsión en cada una de las plataformas que han sufrido una menor variación en sus tiempos de latencia de respuesta ante diferentes contextos (definidos por A, B y C). Notar que la variación de CCS/BB es despreciable, aunque a costa de una latencia muy alta como se ha visto en la Fig. 2. De nuevo, los modos núcleo expulsivo y voluntario aparecen entre los que mejor resultados han obtenido.

Linux sin carga con caches y predicción	
Raspberry (voluntaria)	10117,8ns
BeagleBone(núcleo expulsivo)	26121,25ns
Linux sin carga y sin caches	
Raspberry (voluntaria)	23203,77ns
BeagleBone (N/A)	
Linux sin carga ni predicción de saltos	
Raspberry (núcleo expulsivo)	10303,68ns
BeagleBone (núcleo expulsivo)	27778,86ns

Tabla 6: Datos de la medias de latencia por prueba

Un resultado que se puede extraer de observar ambas gráficas es que el sistema Raspberry tiene una media de latencia de respuesta menor que la BeagleBone, pero su desviación es mayor. Aunque Raspberry y Beaglebone utilizan versiones diferentes del núcleo Linux, el tratamiento de las excepciones es idéntico salvo en lo relativo al al controlador de interrupciones y a las características físicas de los gpios. El acceso al controlador de interrupciones de realiza de la misma forma, pero se trata de un controlador diferente. El interfaz de gestión de gpios también es similar, pero la implementación en cada placa es distinta. En cuanto al comportamiento por modos de expulsión, núcleo expulsivo y voluntaria responden con latencias medias menores.

5.2. Influencia del modelo de expulsión

La Fig. 4 muestra como se comportan los modos de expulsión ante un procesador ocioso o con mucha carga de trabajo. Los modos no tiempo real apenas se ven afectados por ese extra de carga, no así un kernel tiempo real. Como se ha explicado en el apartado 2.2.2, la interrupción es tratada como un *thread* del kernel, por tanto es el planificador tiempo real el que decide cuando entra a ejecutarse.

5.3. Análisis de la variabilidad de la latencia

Para analizar el comportamiento de la latencia en todo el proceso de la gestión de la interrupción, se han realizado varias tomas de medidas en distintos puntos del *handler* de la interrupción y se han encontrado grandes variaciones en la latencia al llegar a ciertos puntos. Estas variaciones se han observado en diferentes pruebas realizadas utilizando diferentes modos de medida. Esta incertidumbre se añade a la ya producida por varios de los

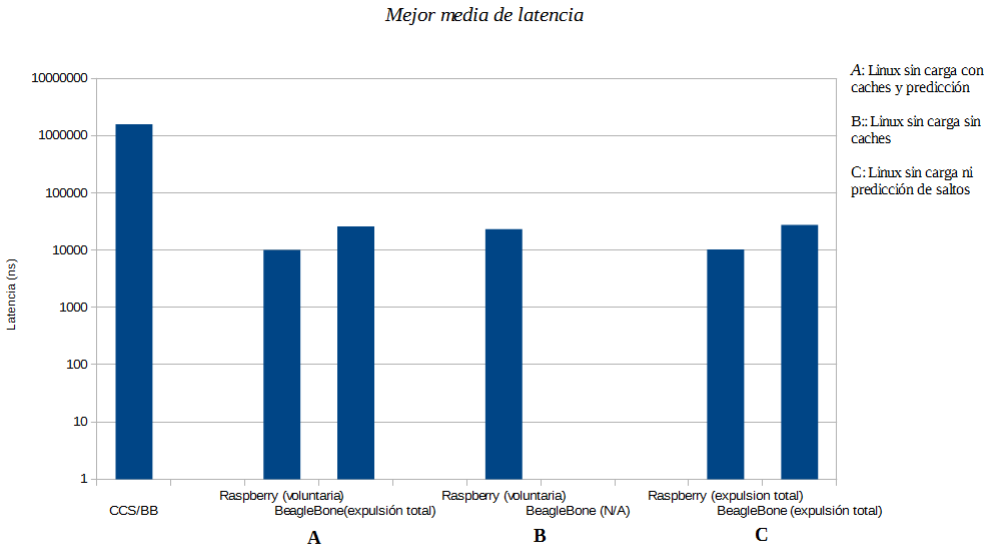

Figura 2: Mejores medias de latencia por prueba (Escala logarítmica)

mecanismos descritos en el apartado 3.

Como ejemplo, se tomará la Fig. 5 que muestra los tiempos acumulados en los puntos de entrada y salida de la cascada de funciones que componen el *handler* en cualquiera de los modos de expulsión del kernel. En este caso en particular se aprecia que la zona azul, etiquetada con el texto vector_IRQ, que se encuentra en la parte inferior, tiene picos de laencia muy altos, aparentemente espúreos. Esta zona corresponde al tiempo que se tarda en saltar a la entrada del vector de interrupciones de la IRQ desde que se genera el evento que la provoca. Una posibilidad que pudiese explicar la generación de estos picos es el interfaz de gpios del sistema, a través de los que se recibe la señal asociada a la IRQ. En cualquier caso se han detectado picos de este tipo en otros puntos del proceso y principalmente en Raspberry. Se han incluido más ejemplos en el capítulo 7.3 del Anexo.

6. Conclusiones

- La programación *baremetal* con rutinas proporcionadas por el fabricante proporciona mayor estabilidad que Linux/ARM, a costa de un rendimiento muy bajo (1.5 ms frente a 10 us de latencia media en Linux)

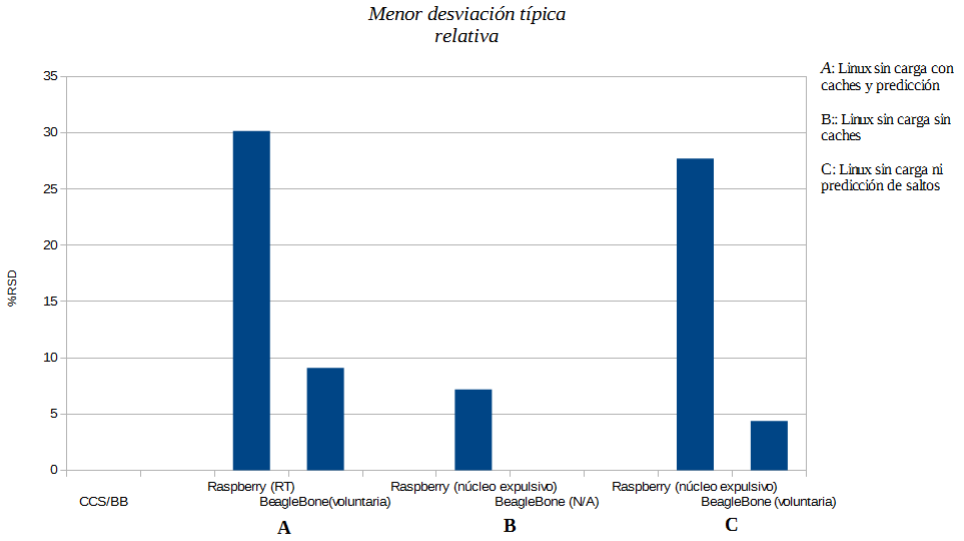

Figura 3: Comparación de los casos con menor variabilidad en la latencia

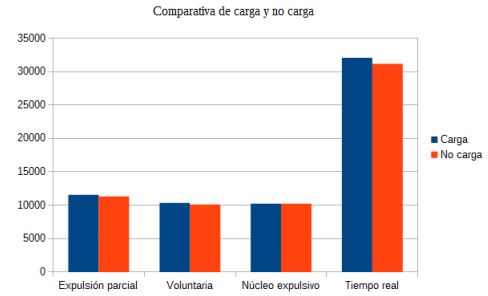

Figura 4: Relación entre pruebas de carga y sin carga (Raspberry)

- La política de expulsión del kernel apenas afecta a la latencia de la interrupción salvo en el modelo de expulsión más agresivo, disponible únicamente en el patch tiempo real.
- Se han detectado partes de los sistemas del entorno de trabajo que afectan a la latencia pero sobre los que no se tiene un control directo. Estas variaciones se han observado en diferentes pruebas realizadas utilizando diferentes modos de medida. Esta incertidumbre se añade a la ya producida por varios de los mecanismos descritos en el apartado 3.

Las placas analizadas son representativas de un tipo de producto muy

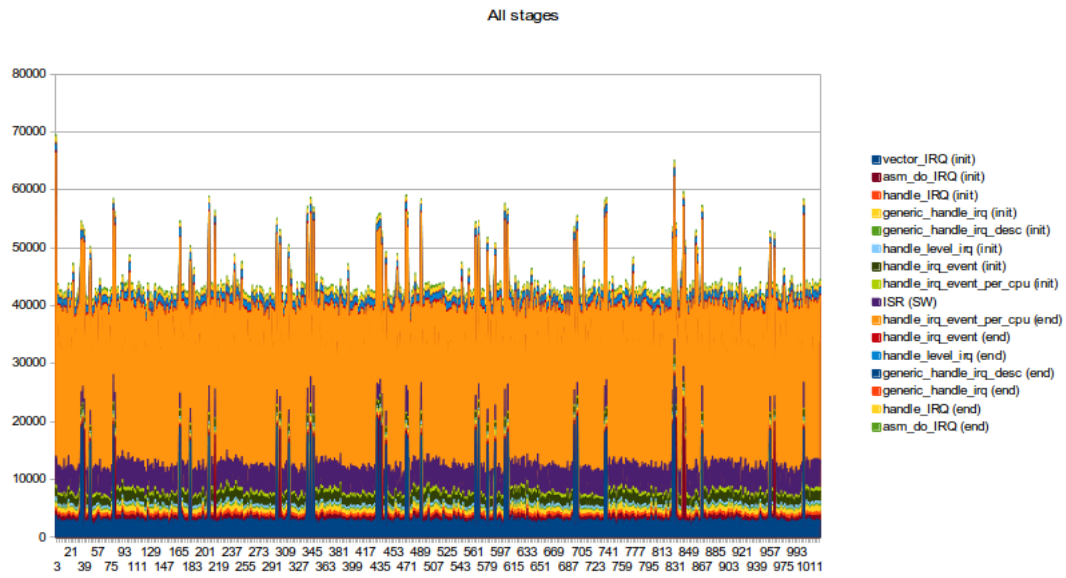

Figura 5: Latencias con carga (1024 muestras, tiempo en ns, Raspberry)

flexible y de bajo coste cada vez más introducido en el mercado. Acompañadas de Linux como sistema operativo, están resultando una opción más que atractiva no sólo para pruebas de concepto sino para desarrollo de soluciones. Pero sus características y en particular los propios SoCs con ARM que incorporan, orientados a aplicaciones, requieren un buen estudio de caracterización previo, antes de decidir su usabilidad para una solución determinada. Así por ejemplo, pueden ser opciones idóneas si el objetivo es implementar un sistema *soft real time* con unos tiempos de respuesta situados en el orden de microsegundos, mientras que si las restricciones son de tiempo real duro, no se pueden exigir tiempos de respuesta menores del orden de milisegundos. En todo caso, en función de la solución objetivo, pueden explotarse optimizaciones ad-hoc, rápidas de conseguir si se tiene un conocimiento exahustivo tanto de las posibilidades de un SoC concreto como de núcleo de Linux.

Referencias

[1] ARM Limited, *ARM1176JZF-S^{TM} Technical Reference Manual*, ARM Limited, 2004.

[2] ARM Limited, *Cortex^{TM}-A8*, ARM Limited, 2006.

[3] Broadcom, *BCM2835 ARM Peripherals*, Broadcom Europe Ltd, 2012.

[4] Sloss, Symes y Wright, *ARM System Developer's Guide*, primera edición, Elsevier, 2004.

[5] Texas Instrument, *AM335x Sitara^{TM} Processors Technical Reference Manual*, Texas Instrument Incorporated, 2011.

7. Anexo

7.1. Top halves versus bottom halves

Una *IRQ handler* tiene que ejecutarse lo antes posible ya que no se van a poder tratar nuevas interrupciones al estar éstas deshabilitadas. Para evitar que no se atiendan interrupciones, dividiremos el tratamiento de la interrupción entre la "top halfz la "bottom half".

- **Top half:** Esta parte comienza nada más recibirse la interrupción y ejecuta solamente el trabajo que se ha de realizar en un tiempo crítico, como el reconocimiento de la interrupción o el reset de hardware.
- **Bottom half:** Aquí se ejecutará código que pueda ser aplazado. Esta parte se ejecutará en un futuro y con las interrupciones habilitadas por lo que puede ser expulsado mientras se está ejecutando. Los mecanismos para poder implementar la bottom half son las softirq, tasklets y las work queues

Nuestro estudio quiere conocer como se comporta la latencia de una interrupción desde el momento en que llega al sistema hasta que el *IRQ handler* comienza a ejecutarse. No obstante, para tener un mayor perspectiva del comportamiento de la gestión de una interrupción en Linux describiremos brevemente los tres mecanismos de la bottom half:

- **Softirq:** Se ejecuta en contexto de interrupción por lo que no pueden ser expulsadas del procesador, excepto por una interrupción que entre. Pueden correr varias softirqs del mismo tipo (misma prioridad) a la vez en distintos procesadores, esto implica que el desarrollador debe incluir mecanismos de control de acceso a recursos compartidos para evitar posibles deadlocks. Se han de definir en tiempo de compilación, el kernel tiene 9 definidas, y el máximo posible es de 32. Estas softirqs son utilizadas para realizar las tareas más críticas.
- **Tasklets:** Este mecanismo bottom half está construido sobre las softirqs. La diferencia entre ambos es que las tasklets tienen un API más amigable para el programador y que no se pueden ejecutar dos instancias de un tasklet del mismo tipo a la vez en varios cores. Esto evita problemas de concurrencia sobre recursos compartidos. Hay dos tipos de tasklets, las HI_SOFTIRQ (mayor prioridad) y las TASKLET_SOFTIRQ.

Tasklet	Priority	Softirq Description
HI_SOFTIRQ	0	High-priority tasklets
TIMER_SOFTIRQ	1	Timers
NET_TX_SOFTIRQ	2	Send network packets
NET_RX_SOFTIRQ	3	Receive network packets
BLOCK_SOFTIRQ	4	Block devices
TASKLET_SOFTIRQ	5	Normal priority tasklets
SCHED_SOFTIRQ	6	planificador
HRTIMERS_SOFTIRQ	7	High-Resolurion timers
RCU_SOFTIRQ	8	RCU locking

Tabla 7: Tipos de softirq

- **Work queues:** Se ejecutan en un thread del kernel en contexto de proceso, por lo que el planificador las puede expulsar. Son útiles en caso de tener código que vaya a bloquearse, debido a un spinlock o un mutex, o que vaya requerir mucho tiempo de ejecución hasta completarse. Puede haber diferentes work queues, cuando se crea

Como hemos visto, las softirq y por extensión las tasklets, son ejecutadas en contexto de interrupción por lo que no pueden ser expulsadas excepto por una interrupción entrante, ni pueden invocar directamente al planificador (*direct invocation*) sino que han de activarlo inicializando a 1 la variable *need_resched* (invocación retardada o *lazy invocation*). El planificador invocado mediante invocación retardada se activa al testear el valor de *need_resched*. Esto se hace en un código que se ejecuta al regreso de cualquier excepción, síncrona o asíncrona (interrupciones) del sistema. Las softirq pueden reactivarse a sí mismas. Cuando esto se detecta, para evitar la degradación en el rendimiento del sistema, el planificador no gestiona softirqs reactivadas. Cuando existen muchas softirq pendientes de ejecución, se despierta un thread de kernel específico (*ksoftirqd*), una instancia del mismo por procesador, que ejecuta las softirqs pendientes. Este thread se ejecuta con la prioridad más baja posible a fin de no retrasar la ejecución del resto de tareas.

7.2. Controlador de de interrupciones vectorizados

Estos dispositivos proporcionan una forma eficiente de reconocer fuentes de interrupción (las *Int soource X* que aparecen en la imagen 6) y asociarlas a su ISR correspondiente. De esta manera, se evita el proceso de gestión del *handler* de la interrupción.

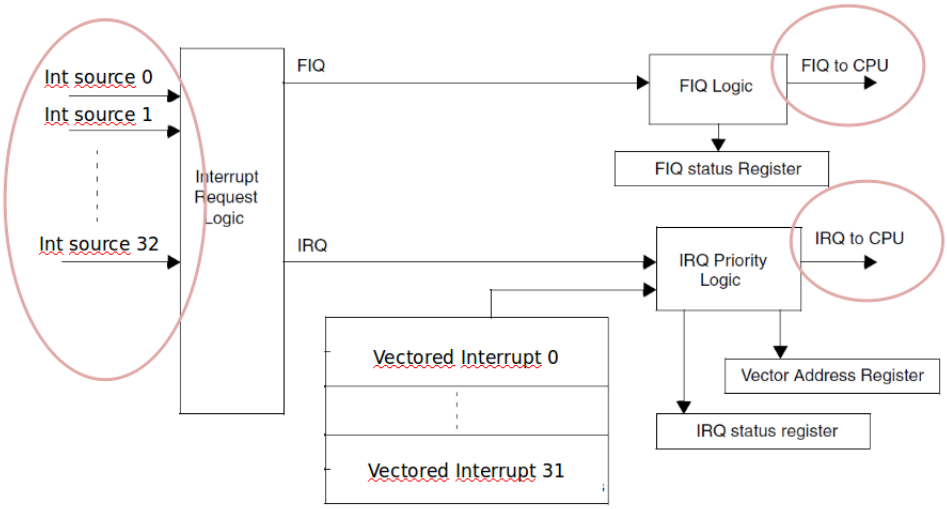

Figura 6: Funcionamiento de un controlador de interrupciones vectorizado (VIC)

El tener diversas entradas de interrupción permite asociar una por fuente de interrupción, evitando que varias la misma entrada tengan que realizar un proceso de *polling* en un registro para saber cual de ellas generó la interrupción. Además pueden llegar a incluir otras funcionalidades como prioridad de interrupciones o asociación de una fuente IRQ a la FIQ

7.3. Medidas que incluyen picos importantes de latencia

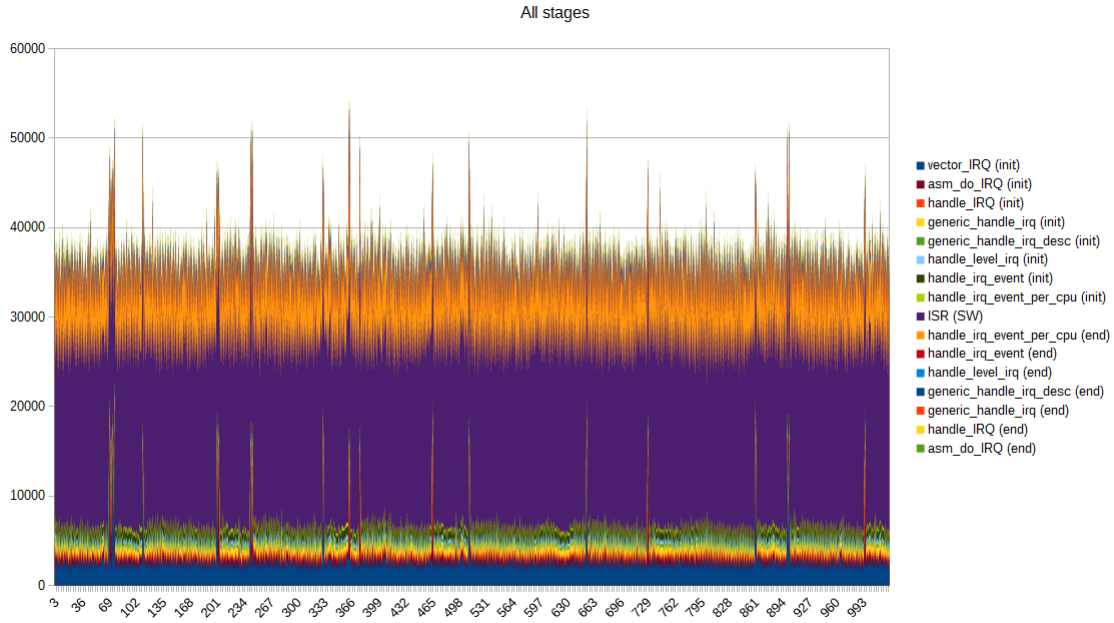

Figura 7: Medidas con picos de latencia (BeagleBone, expulsión total, no predicción de saltos

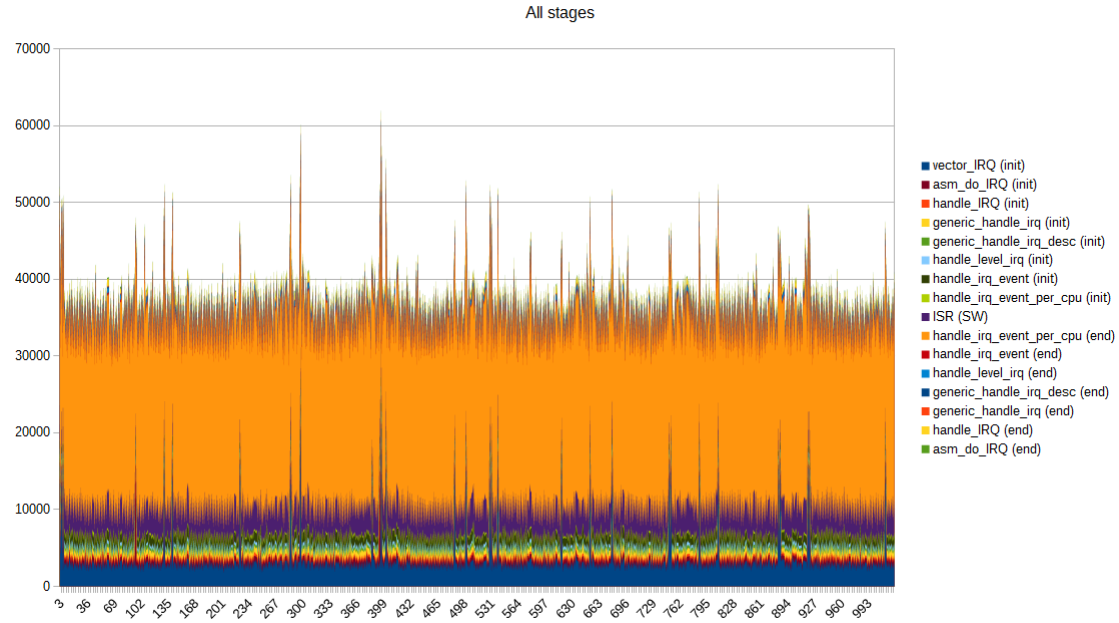

Figura 8: Medidas con picos de latencia (Raspberry, voluntaria, no predicción de saltos

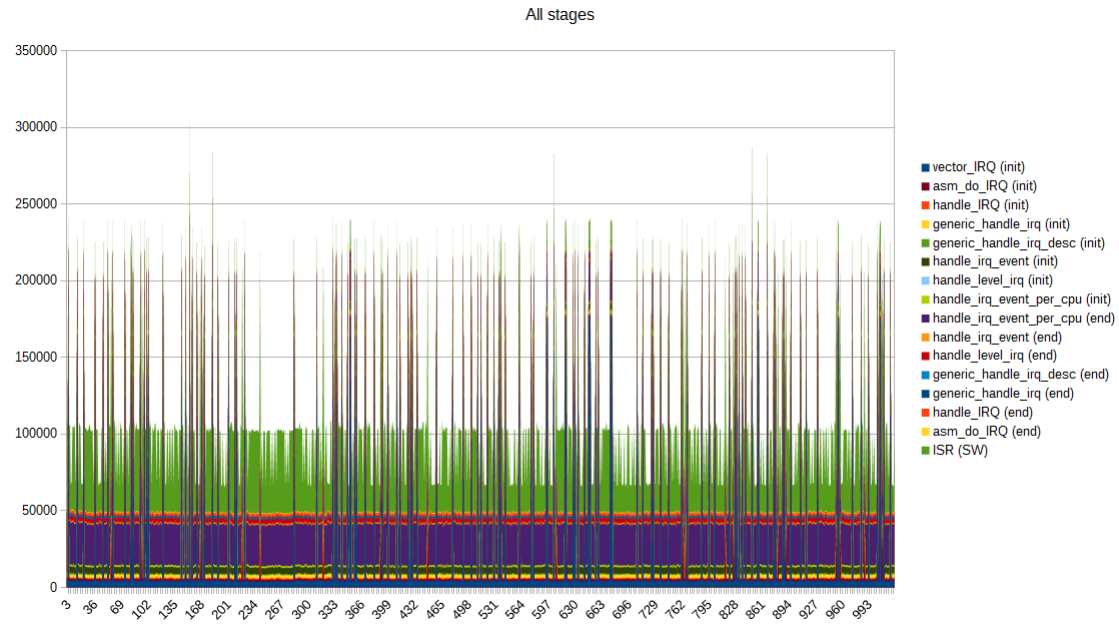

Figura 9: Medidas con picos de latencia (Raspberry, tiempo real, carga

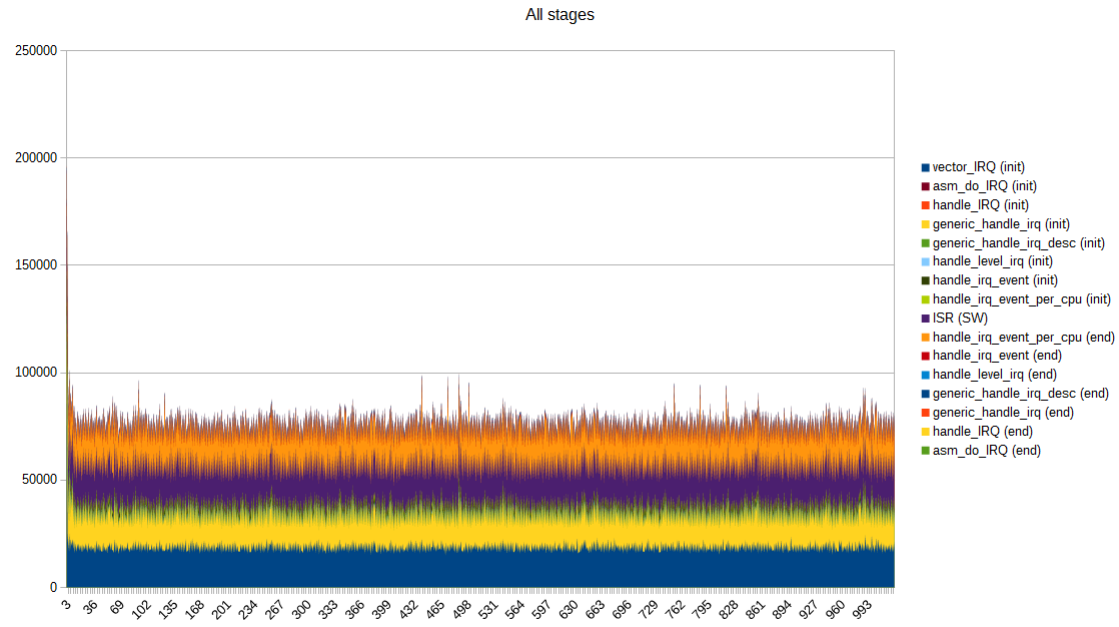

Figura 10: Medidas con picos de latencia (BeagleBone, expulsión parcial, carga