Comunicaciones Multigigabit para SoC basado en FPGA

Manuel Sánchez Gómez, Raúl Mateos Gil
Departamento de Electrónica,
Universidad de Alcalá
{ manuel.sanchez; raul}@depeca.uah.es

Luis Medina Valdés
CAEND/IAI
Consejo Superior de Investigaciones Científicas
luis.medina@csic.es

Resumen— **Este artículo muestra una solución para comunicaciones mutigigabit basadas en el protocolo Aurora utilizado en SoC para aplicaciones de imágenes médica por ultrasonidos. Esta solución consta de un IP que utiliza canales DMA para mover eficientemente los datos a transmitir/recibir desde/hacia la memoria de usuario. Este IP se suministra junto con un driver de dispositivo que proporciona una sencilla API de usuario para facilitar el desarrollo de aplicaciones. La integración del driver con el resto del sistema está completamente resuelta extrayendo automáticamente toda la información arquitectural necesaria para adaptarse a las características de cada diseño planteado.**

Palabras clave—**FPGA, System on Chip, comunicaciones multigigabit.**

I. INTRODUCCIÓN

EN aplicaciones de imagen médica mediante ultrasonidos es una práctica común el uso de FPGAs debido a la gran flexibilidad que ofrecen, permitiendo reducir sustancialmente el tamaño y consumo de este tipo de equipos. Estos diseños suelen utilizar una aproximación SoC (*System on Chip*), integrando en una sola FPGA la cadena de procesamiento de la señal ultrasónica junto con un microprocesador que se encarga de gestionar las comunicaciones con un ordenador remoto en el que se visualizan las imágenes a partir de la información recibida desde el SoC. Para esta comunicación se suelen utilizar enlaces USB2 o Ethernet (100/1000 Mb), implementados mediante controladores externos a la FPGA [1].

No obstante, a medida que aumenta el grado de exigencia de estas aplicaciones el volumen de datos a tratar crece exponencialmente. Por ejemplo, los sistemas de tomografía por ultrasonidos pueden utilizar arrays de 256, 512 o incluso 1024 elementos. La aproximación más habitual para adquirir y procesar tal cantidad de canales consiste en utilizar una arquitectura segmentada compuesta por un conjunto de módulos de procesamiento basados en FPGAs (véase Fig. 1). Estos módulos se comunican mediante enlaces paralelo punto a punto de alta velocidad (400 MT/s).Al final de la cadena aparece un módulo que se encarga de transferir estos datos al ordenador remoto.

Claramente, con este volumen de datos resulta imposible seguir utilizando los enlaces de comunicación típicos anteriormente citados. Las FPGAs actuales integran transceivers multigigabit (MGT) que facilitan la implementación de enlaces de comunicación serie con tasas de transferencia por encima 1 Gb/s. De hecho han sido utilizados para implementar la capa física de un gran número de protocolos, p.e. SATA, Serial RapidIO, Infiniband, etc. [2].

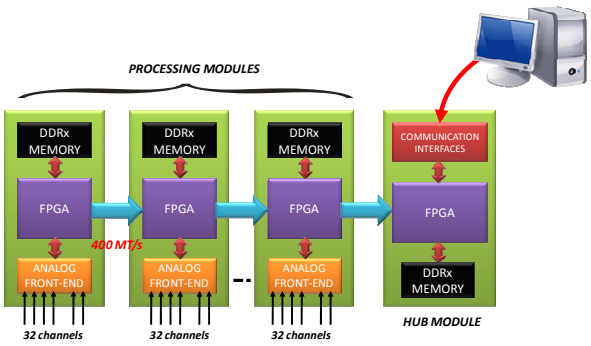

Fig. 1. Sistema de tomografía por ultrasonidos.

La solución más adecuada para crear el enlace de comunicaciones es optar por un protocolo estándar, que presente una elevada tasa de transferencia y sea sencillo. Uno de estos protocolos estándar es Ethernet 10G. No obstante, esta opción requiere del uso de chips externos para implementar el controlador de capa física. Por otra parte, la sobrecarga que introduce la implementación software del protocolo TCP/IP reduce drásticamente la velocidad de transferencia.

Una alternativa muy interesante es el protocolo Aurora de Xilinx [3]. Se trata de un protocolo sencillo, ligero y que proporciona tasas de transferencia de hasta 3.2 Gbps. Existen disponibles cores Aurora parametrizables pero su integración en sistemas SoCs no está resuelta. Uno de los problemas que se plantea a la hora de su integración en un SoC es el movimiento eficiente de los datos a transmitir y/o recibir.

En este artículo se muestra el desarrollo de un IP para comunicaciones mutigigabit basado en el protocolo Aurora integrable en cualquier diseño EDK. Dicho core utiliza canales DMA para mover los datos desde/hacia la memoria de usuario. En lo referente a los aspectos software se ha desarrollado un driver de dispositivo que proporciona una sencilla API de usuario que facilita el desarrollo de aplicaciones. Su integración en EDK está completamente resuelta extrayendo automáticamente toda la información arquitectural necesaria para adaptarse a las características de cada diseño.

El artículo está organizado tal y como se describe a continuación. En el apartado 2 se describe brevemente los fundamentos de Aurora. En el apartado 3 se abordan todos los aspectos referentes al diseño del IP de comunicaciones y su integración en el SoC. En el apartado 4 se describe el driver de dispositivo desarrollado y su uso en una aplicación de usuario. El análisis de las prestaciones del sistema y los resultados obtenidos se incluyen en el apartado 5. El artículo concluye con un apartado de conclusiones y trabajos futuros.

II. PROTOCOLO AURORA

Aurora es un protocolo ligero de comunicaciones serie de alta velocidad (multigigabit) para conexiones punto a punto. Es escalable pudiendo hacer uso de una o varias pistas o *lanes*, tal y como se muestra en la Fig. 2, cada una de las cuales tiene asociado un MGT (denominados GTP en la familia V5). Admite tanto comunicaciones dúplex como *half-duplex*. Hay dos versiones de Aurora dependiendo de la codificación de línea utilizada: 8B/10B y 64B/66B. Los cores de Aurora se pueden generar automáticamente mediante la herramienta coregen. En el diseño planteado se utilizarán cores Aurora 8B/10B de una sola pista. El core Aurora lleva a cabo automáticamente la inicialización del canal cuando detecta la conexión del agente interlocutor. Una vez inicializado el canal, la aplicación puede proceder al envío de datos en forma de tramas. Las tramas pueden tener un tamaño arbitrario e interrumpirse en cualquier momento. Cuando no se transmiten datos por el canal Aurora envía códigos *idle* para mantener el enlace activo.

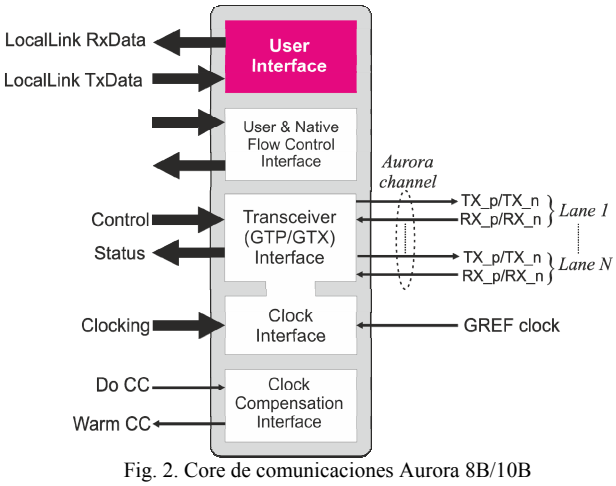

Fig. 2. Core de comunicaciones Aurora 8B/10B

En Fig. 2 se muestra el diagrama de bloques del core generado y su interface externo. El core admite dos modos interfaces de usuario: *streaming* and *framing*. En el modo *streaming* los datos aparecen en un flujo continuo mientras que en el modo *framing* se agrupan en forma de tramas, siendo este último el utilizado en este diseño. El interface *framing* utiliza el protocolo LocalLink para la transferencia de datos. LocalLink (LL) [4] es un protocolo síncrono tipo stream para comunicaciones unidireccionales punto a punto. La anchura puede ser configurable. Tal y como muestra Fig. 3 cada uno de los agentes (iniciador y destinatario) dispone de su propia señal de validación (handshaking) para controlar la temporización de la transferencia de datos. Las tramas LL constan de tres secciones: una cabecera, el contenido de datos (payload) y un pie. La cabecera y el pie son opcionales y se utilizan para transmitir información auxiliar o de control (metainfo). La longitud de estas secciones se delimita mediante una serie de señales de control adicionales. Cuando el número de bytes a transferir no es un múltiplo entero de la anchura de LL el último dato del payload puede estar incompleto. En este caso se utiliza la señal `rem` para indicar el número de bytes válidos del último dato. Este resto se puede expresar de forma decodificada (un bit por cada byte) o como un código binario (índice del último byte válido del bus).

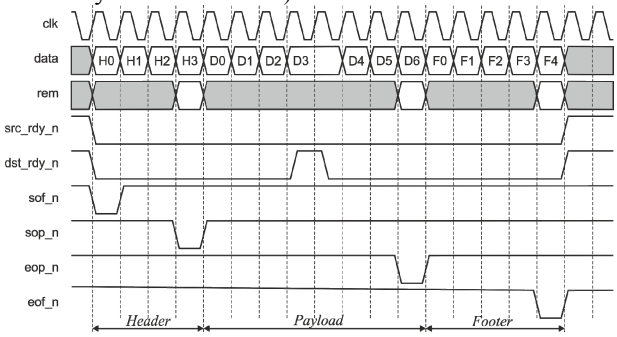

Fig. 3. Estructura de una trama localink.

Opcionalmente, Aurora puede utilizar control de flujo, admitiendo dos tipos: nativo y de usuario. El control de flujo de usuario se utiliza para enviar mensajes cortos con una alta prioridad a través del canal. La latencia de estos mensajes es mucho menor que la de los datos convencionales, utilizándose habitualmente para labores de control. El control de flujo nativo se utiliza para regular la tasa de los datos entrantes. Este tipo de mensajes se le envía al otro interlocutor para que reduzca la tasa de datos enviados evitando de esta forma situaciones de rebose. En el diseño no se ha utilizado ningún tipo de control de flujo.

El bloque de interface de los transceivers incluye puertos de I/P asociados a los GTP/GTX utilizados para implementar el enlace. Además dispone de varias señales de control y estado para la gestión del enlace. Las señales de control se utilizan para realizar operaciones tales como la inicialización del core (reset), habilitación del modo bucle (loopback), corrección de reloj, paso a bajo consumo, etc. Las señales de estado proporcionan información sobre el estado del canal, aparición de errores, etc. Este bloque está íntimamente relacionado con el bloque de generación de reloj. Recibe el reloj de referencia utilizado por los transceivers y genera un reloj de salida a partir del cual se deben derivar externamente (mediante un DCM) todos los relojes utilizados por la interface de usuario.

En un sistema Aurora, ambos interlocutores deben utilizar reloj de referencia con la misma frecuencia. No obstante, cada agente utiliza su propia señal de referencia por lo que pueden aparecer desviaciones respecto al valor nominal de la citada frecuencia. El bloque de compensación de reloj se utiliza para corregir estas desviaciones (hasta un máximo ±100 ppm), garantizando el correcto funcionamiento de las comunicaciones. Para ello se envían secuencias de entrenamiento a través del canal al activar la señal `DO_CC`. Durante este tiempo no se envían por el canal ni datos ni mensajes de control de flujo. La señal `WARM_CC` se utiliza para evitar la generación de mensajes de control de flujo próximos a estas secuencias de entrenamiento.

Coregen genera el core de Aurora junto con un diseño de referencia en VHDL en el que se incluyen los módulos necesarios para generar los relojes del sistema y controlar la compensación de reloj. Partiendo de esta

infraestructura básica se procederá al diseño del IP de comunicaciones para el SoC, tal y como se describe en el siguiente apartado.

III. CORE DESARROLLADO.

Para posibilitar la integración de las comunicaciones Aurora en un diseño de un SoC se ha desarrollado un IP denominado `xps_aurora_dma`. Este IP encapsula el core Aurora generado por coregen, adaptando su conexión con los restantes elementos del SoC. En la Fig. 4 se muestra la arquitectura de un SoC basado en Microblaze que hace uso del IP desarrollado. Para garantizar una elevada velocidad de transferencia el movimiento de datos hacia/desde el core Aurora se realizará mediante DMA. Para ello `xps_aurora_dma` implementa un cliente DMA que se conecta a uno de los puertos del controlador de memoria MPMC [5]. Por otro lado el acceso a los registros de control y estado del IP se realiza a través de un interface PLB.

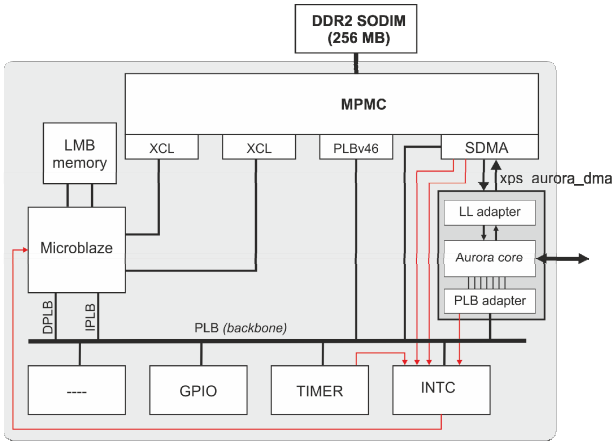

Fig. 4. Integración del IP en un SoC

Tal y como se muestra en la Fig. 5 el IP desarrollado consta de tres módulos. El módulo principal, denominado `xps_aurora_dma_core`, integra el core de Aurora. En color aparecen los módulos principales que componen el core generado por Coregen. Además del propio core Aurora aparecen dos bloques auxiliares. El primero de ellos es el bloque de compensación de reloj. Este módulo genera periódicamente secuencias de entrenamiento para asegurar el funcionamiento de las comunicaciones frente a las diferencias de la frecuencia de reloj de los agentes involucrados en la comunicación. El segundo bloque genera el reloj de la interface de usuario del core (canales local link). Dado que el core y el resto del SoC utilizan relojes distintos se han incorporado sendas FIFOs asíncronas (una por cada sentido de la comunicación) para desacoplar ambos dominios de reloj.

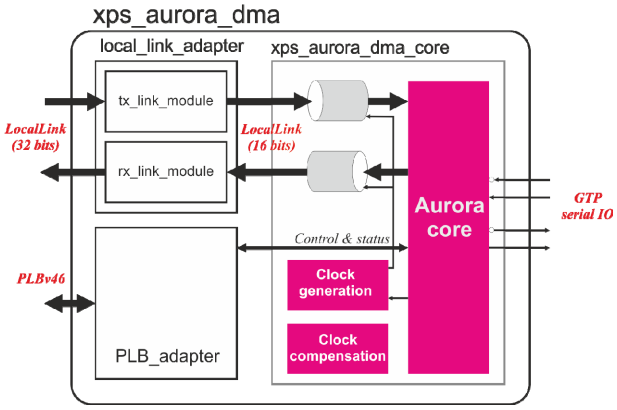

Fig. 5. Diagrama de bloques del IP desarrollado.

El módulo `local_link_adapter` implementa un cliente DMA cuya función principal es realizar la conversión de formato entre las tramas LL utilizadas por el controlador DMA y el core de Aurora. Aunque ambos agentes, DMA y Aurora, utilizan LL como protocolo de comunicaciones, el formato de las tramas en ambos casos presenta ciertas diferencias. En primer lugar la anchura de los canales LL del controlador DMA son de 32 bits, mientras que Aurora utiliza buses de 16 bits. Esto obliga a realizar una conversión de anchura en el citado módulo. Además el formato de la señal de resto es distinto (decodificado para DMA y binario para Aurora). En lo referente al formato de la trama, Aurora sólo utiliza la sección de payload, mientras que el DMA utiliza la cabecera y el pie para incorporar cierta información adicional (p.e. número de bytes transferidos). Finalmente ambos agentes (DMA y Aurora) utilizan dominios de reloj distintos.

El cliente DMA consta de dos módulos, uno de transmisión y otro de recepción. El módulo de transmisión (`tx_link_module`) recibe las tramas LL procedentes del controlador DMA, extrae los datos de la trama (descartando la cabecera y el pie) y los transmite al core de Aurora, adaptando la anchura de la trama de salida a 16 bits (*downsizing*). Por su parte, el módulo de recepción (`rx_link_module`) recibe las tramas LL procedentes del core de Aurora, empaqueta los datos que contienen en $word_{32}$ (*upsizing*) y añade a la trama de salida una cabecera y un pie. La cabecera tiene una longitud de una $word_{32}$, siendo su contenido irrelevante. El pie tiene una longitud de 8 $word_{32}$ y la última debe contener el tamaño del payload en bytes.

Finalmente el módulo `plb_adapter` es un esclavo PLB que incorpora una serie de registros de control y estado mediante los cuales el procesador puede configurar y supervisar los aspectos generales del funcionamiento de las comunicaciones Aurora: reset del core, paso a modo bajo consumo, detección del estado del enlace, etc.

IV. ASPECTOS SOFTWARE

Con objeto de facilitar el uso del IP descrito es necesario proporcionar al usuario una sencilla API para el envío y recepción de bloques de datos. Esto hace necesario el desarrollo de un driver de dispositivo que gestione su operación y oculte los detalles de su funcionamiento interno. Este driver ha de operar de forma cooperativa con otros drivers de dispositivos involucrados en el envío y recepción de datos, como son el controlador DMA y de interrupciones. Todos estos drivers forman parte del BSP (Board Support Package) del SoC.

Al contrario que ocurre con los SoCs convencionales, los cuales presentan una arquitectura fija, el alto grado de configurabilidad que presentan los SoCs basados en FPGAs suponen una dificultad añadida en el desarrollo del BSP. En el caso del IP desarrollado esto no sólo se limita a aspectos propios del IP (p.e. posición dentro del mapa de memoria que se incluye en el fichero `xparameters.h` generado por EDK), sino que es preciso considerar también otros elementos a los que se conecta. Tal y como muestra la figura 1, el IP puede conectarse a cualquiera de los 8 controladores DMA que incorpora el MPMC, ubicándose los registros de control de cada uno de ellos en distintas posiciones del mapa de memoria. Además, las interrupciones generadas por el controlador DMA empleado y por el propio IP se pueden conectar a distintas entradas del controlador de interrupciones lo cual influye en el tratamiento a aplicar. Toda esta información estructural está contenida en el fichero de especificación hardware MHS y debe ser tomada en consideración durante el desarrollo del driver. Dado que todos estos aspectos pueden variar de un diseño a otro es preciso proporcionar un método que evite tener que reescribir el código fuente del driver para cada uno de los diseños realizados.

En la Fig. 6 se muestra la solución aportada a esta problemática para automatizar la generación del driver. Cada vez que se ejecuta la herramienta de generación de librerías `libgen` EDK mantiene una base de datos con la información estructural del diseño, permitiendo acceder a ella mediante una API TCL [6]. Además del fichero MDD, todo driver de dispositivo puede tener asociado un script TCL que `libgen` ejecuta durante la generación del BSP y desde el que se puede acceder a la citada base de datos. Los ficheros fuente que constituyen el driver se dividen en dos grupos. El primero de ellos esta formado por los ficheros almacenados en el directorio `src` del driver junto con el fichero Makefile utilizado para compilarlos. Estos ficheros proporcionan la funcionalidad del driver y no varían con el diseño. El segundo grupo está formado por uno o varios ficheros fuente que se generan automáticamente desde el script TCL para cada diseño. Para ello el script TCL debe incluir un procedimiento llamado `generate` desde el que se accederá a la base de datos. Los ficheros C generados contienen básicamente una estructura de datos con la información de configuración del IP (valores de parámetros) y su conectividad. Esta estructura de datos se utilizará para adecuar el funcionamiento del driver a cada diseño en particular.

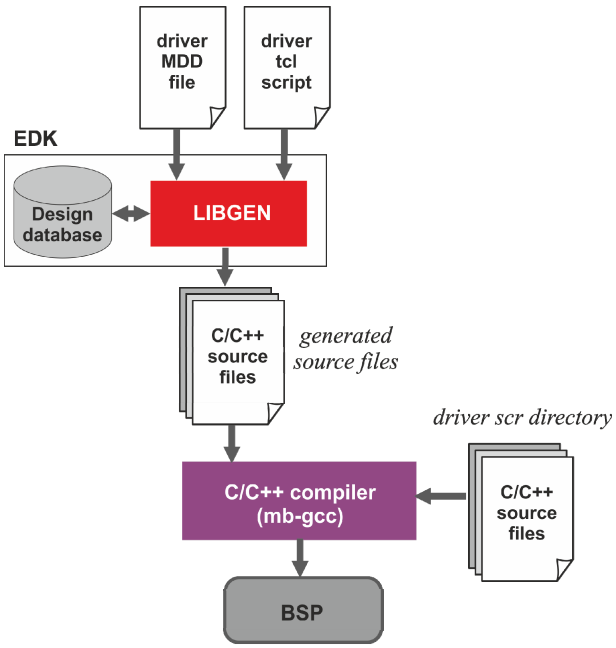

Fig. 6. Automatización de la generación del driver de dispositivo.

El API del driver está compuesto por las siguientes funciones:

```
XStatus   xps_aurora_Initialize(xps_aurora
*dev, uint16 devID);
int  xps_aurora_dma_send(xps_aurora  *dev,
const void *data, size_t size);
int  xps_aurora_dma_recv(xps_aurora  *dev,
void *data, size_t size);
```

La función `xps_aurora_Initialize` lleva a cabo la inicialización del driver. En primer lugar configura el IP fijando los valores adecuados en sus registros de control y espera a detectar que el enlace Aurora está operativo. A continuación inicializa el controlador DMA utilizado para la transferencia de datos a/desde la memoria, creando los anillos de descriptores asociados a los buffers que contendrán los datos a transmitir y recibir. Finalmente instala los gestores de interrupción correspondientes al controlador de DMA (una para Tx y otra para Rx) y la del propio IP. Esta última se utiliza para detectar situaciones anómalas como la ruptura de comunicación en el enlace Aurora.

El tratamiento aplicado en las transferencias DMA difiere sustancialmente para la transmisión y la recepción. La transmisión tiene un carácter síncrono respecto a la ejecución de la aplicación de usuario ya que la transferencia DMA sólo se iniciará al realizar una llamada a la función `xps_aurora_dma_send`, siendo conocido en ese momento el tamaño del buffer a transmitir. En ese momento la función reserva (`malloc`) un buffer en el que copiar el bloque de datos a transmitir, toma un descriptor libre del anillo de descriptores, lo inicializa para que apunte al buffer reservado y se lo entrega al controlador DMA para que proceda a su envío. La entrega del descriptor no supone su transmisión inmediata, sino que este se encola al final del anillo de descriptores. Al concluir la transmisión del buffer se dispara la interrupción de transmisión la cual se encarga de liberar el descriptor asociado y el buffer reservado previamente.

Por otro lado, la recepción presenta un carácter asíncrono respecto a la aplicación de usuario ya que los paquetes pueden llegar en cualquier momento, no siendo conocido su tamaño hasta el momento en que finaliza su recepción. Esto obliga a utilizar una estrategia distinta, reservando los buffers por adelantado. Para evitar la pérdida de datos, durante la inicialización del driver se reserva un buffer de memoria para cada uno de los descriptores del anillo de recepción y se le entregan todos ellos al controlador. Estos buffers se dimensionan a un tamaño máximo que depende de cada aplicación en particular. Tal y como se muestra en la Fig. 7, al comenzar la recepción de un paquete el controlador DMA toma el primer descriptor disponible del anillo y almacena los datos recibidos en el buffer asociado. Al concluir la recepción del paquete el gestor de la interrupción de recepción de DMA restituye los valores por defecto del descriptor, le reserva un nuevo buffer de datos y se lo entrega de nuevo al controlador para que lo vuelva utilizar, esta vez ocupando el final de la cola. Sin embargo, el buffer originalmente asignado al descriptor no se libera, sino que se introduce el puntero asociado en una cola (FIFO) software. Cuando la aplicación de usuario desea leer un nuevo paquete recibido realiza una llamada a la función `xps_aurora_dma_recv`, la cual extrae el siguiente puntero de la cola software, copia el contenido del buffer DMA sobre el buffer de usuario que recibe como parámetro y libera el buffer DMA una vez que se ha consumido todo su contenido.

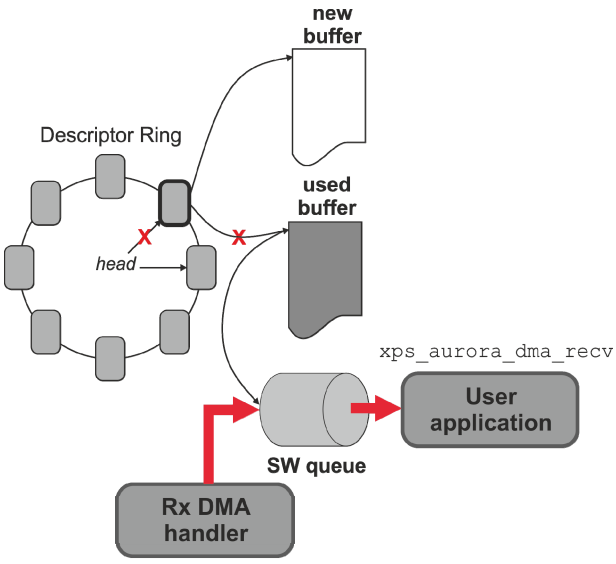

Fig. 7. Gestión software de recepción de paquetes vía DMA.

V. RESULTADOS EXPERIMENTALES

El diseño realizado se ha validado utilizando sendas tarjetas XUP-V5, una operando como transmisora y otra como receptora. La tarjeta XUP-V5 está basada en una FPGA Virtex 5 LX110T y dispone de varios tipos de conectores para acceder a los MGT que integra la FPGA. En los ensayos realizados se han considerado tanto las comunicaciones a través de cable de cobre (conectores SMA) como por fibra óptica (conector SFP). En el caso de las comunicaciones por cable de cobre los MGTs operan a la máxima frecuencia de operación (3.1 Gbps) obteniéndose unas tasas reales en modo sostenido de 214 MBytes/s. Esto supone una eficiencia del 69% sobre la velocidad de pico.

En el caso de las comunicaciones ópticas la frecuencia de operación de Aurora se redujo a 2.1 Gbps debido a la limitación que presentan los transceiver SFP utilizados (2.125 Gbps). En este caso la velocidad de transferencia en modo sostenido que se obtuvo fue de 183 MBytes/s, lo que supone un 87% de la velocidad de pico.

Lógicamente la velocidad de transferencia está relacionada con el tamaño del paquete empleado. Fig. 8 muestra los resultados obtenidos de la evolución de la velocidad de transferencia en función del tamaño del paquete para el caso de las comunicaciones SFP. Como se aprecia, para tamaños de paquete pequeños (<500 bytes) el rendimiento es bajo. Esto se debe a la sobrecarga introducida por la atención de las interrupciones (aspecto SW) y penalización que supone para el DMA el acceso a los descriptores (aspecto HW). A partir de tamaños de 2 KB la velocidad se estabiliza.

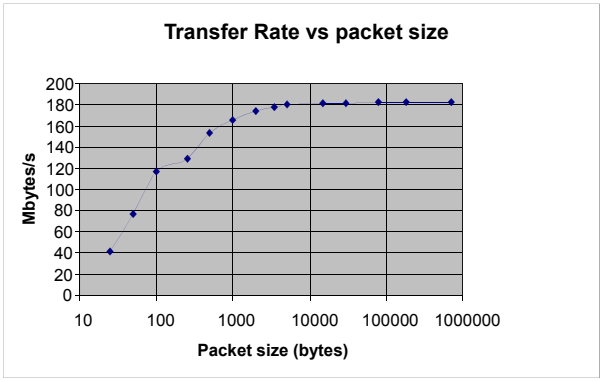

Fig. 8. Velocidad de transferencia en función del tamaño del paquete.

Otro aspecto a valorar del diseño es el consumo de recursos lógicos. En la siguiente tabla se muestra el consumo de recursos del SoC completo y del IP desarrollado en particular. Las cifras que aparecen entre paréntesis representan el porcentaje sobre el total de los recursos disponibles en la FPGA.

TABLA 1. CONSUMO DE RECURSOS LÓGICOS

	Slices	BRAMs	GTP dual
SoC	4841(28%)	36 (24%)	1(12%)
xps_aurora_dma	701(4 %)	3 (2%)	1(12%)

Como se puede apreciar el IP desarrollado resulta muy eficiente en términos de consumo de recursos. Si se compara con otros IPs de comunicaciones de alta velocidad como la MAC Ethernet HW (Slices=880, BRAM=4), se observa que el IP para comunicaciones basadas en Aurora presenta un consumo de recursos similar pero ofrece unas tasas de transferencia muy superiores.

VI. CONCLUSIONES

En este trabajo se ha presentado una solución completa de comunicaciones de alta velocidad basadas en Aurora para SoCs basados en FPGAs. Dicha solución consta de

un IP que incluye un cliente DMA para optimizar la transferencia de datos entre la memoria del SoC y el core de Aurora. El IP desarrollado se complementa con su correspondiente driver de dispositivo. Este driver proporciona una sencilla API lo que facilita enormemente el desarrollo de aplicaciones de usuario. El driver está completamente integrado en EDK obteniendo automáticamente toda la información necesaria para el manejo del IP, lo que garantiza su portabilidad en distintos diseños sin necesidad de reescribir parte de su código fuente. La solución completa permite obtener elevadas tasas de transferencia de hasta 214 MB/s, con un consumo de recursos lógicos sumamente reducido.

AGRADECIMIENTOS

El presente trabajo ha sido financiado mediante el proyecto ARTEMIS 52009/DPI-1802 de la Comunidad de Madrid.

REFERENCIAS

[1] F. M. Sánchez, J. F. Cruza, R. Mateos, C. Fritsch, R. Bueno, “Sistema ultrasónico multicanal en un chip”, XI Jornadas de Computación Reconfigurable y Aplicaciones (JCRA'11), La Laguna, 7-9 Sept. 2011.

[2] A. Athavale, C. Christensen "High-Speed Serial I/O Made Simple", 2005.

[3] Xilinx, "Aurora 8B/10B Protocol Specification. v2.2". 2010.

[4] Xilinx, "LocalLink reference guide". 2010.

[5] Xilinx, "LogiCORE IP Multi-Port Memory Controller (MPMC) (v6.04.a)". 2011.

[6] Xilinx, "Embedded System Tools Reference Manual. EDK 13.2". 2011

[7] Xilinx, "Platform Specification Format Reference Manual. EDK 13.2". 2011