UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DISCA
DEPARTAMENTO DE INFORMÁTICA DE SISTEMAS Y COMPUTADORES

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Diseño e Implementación de Soporte de Calidad de Servicio en Arquitecturas Multinúcleo

TRABAJO FINAL DE MÁSTER

Máster en Ingeniería de Computadores y Redes

Autor:
Tomás Picornell Sanjuan

Tutor:
José Flich Cardo

Valencia, Julio de 2016

Resumen

En este trabajo se diseña e implementa una red en el chip con soporte de calidad de servicio. La red es reconfigurable y permite tanto la instanciación de un número variable de unidades funcionales y recursos, como la obtención de diferentes configuraciones, cada una con una relación prestaciones/recursos distinta. La red se encuentra totalmente integrada en la arquitectura multinúcleo PEAK desarrollada en el Grupo de Arquitecturas Paralelas (GAP) del Departamento de Informática de Sistemas y Computadores (DISCA) de la Universitat Politècnica de València (UPV). Esta arquitectura esta englobada en el proyecto europeo MANGO (`http://www.mango-project.eu`). La red desrrollada con soporte a calidad de servicio será utilizada en dicho proyecto para garantizar anchos de banda y latencias a diferentes aplicaciones con requisitos temporales.

La red implementa redes virtuales las cuales contienen a su vez canales virtuales. Estas redes virtuales permiten separar el tráfico de forma lógica y por lo tanto tiene la capacidad de reservar anchos de banda a las diferentes redes virtuales con el fin de garantizar calidad de servicio.

El trabajo incluye el diseño de tests de prueba y ejecuciones reales con toda la arquitectura funcionando sobre un sistema operativo propio que verifican y validan cada componente así como y las diferentes configuraciones finales de la red de interconexión. Por otro lado, se ha sintetizado el conmutador para cada una de las configuraciones con el fin de obtener los recursos que necesita para su implementación en un sistema FPGA. A lo largo del desarrollo del trabajo se han utilizado herramientas comerciales como Vivado de Xilinx, software de control de versiones (Git) u otras propias como el sistema operativo PEAKos.

Palabras clave: NoC, calidad de servicio, FPGA, sistemas multinúcleo, canales virtuales, redes virtuales, configurable.

Resum

En aquest treball es dissenya i implementa una xarxa en el chip amb suport de qualitat de servei. La xarxa es reconfigurable y permet tant la instanciació dún nombre variable d'unitats funcionals i recursos com lóbtenció de diferents configuracions, cadascuna amb una relació prestacions/recursos diferent. La xarxa s'encontra totalment integrada en l'arquitectura multinúcli PEAK desenvolupada en el Grup d'Arquitectures Paral·leles (GAP) del Departament d'Informàtica de Sistemes i Computadors (DISCA) de la Universitat Politècnica de València (UPV).

La xarxa inclou xarxes virtuals les cuals contenen a la vegada canals virtuals. Estes xarxes virtuals permenten separar el tràfic de forma lògica y per lo tant té la capacitat de reservar amplis de banda a cadascuna de les xarxes virtuals amb la finalitat de garantizar qualitat de servei.

El treball inclou el diseny de tests de prova i ejecucions reals amb tota l'arquitectura funcionant sobre un sistema operatiu propi que verifiquen i validen cada component i les diferents configuracions finals de la xarxa d'ínterconexió. D'altra banda s'ha sintetitzat cadascun dels components del conmutador per a cadascuna de les configuracions amb la finalitat d'obtenir els recursos que necessita per a la seua implementació en un sistema FPGA. Al llarg del desenvolupament del treball s'han fet servir diverses ferramentes comercials com ara Vivado de Xilinx, programari de control de versions (Git) o altres propietaries com es el sistema operatiu PEAKos.

Paraules clau: NoC, qualitat de servei, FPGA, sistemes multinúcli, canals virtuals, xarxes virtuals, configurable.

Abstract

This project involves the design and implementation of a network on chip with quality of service support. The network is configurable, allowing a variable number of resources and functional units. Different configurations can be created, each with a different performance/resource ratio. This network is fully integrated into the PEAK multicore architecture developed by the Grupo de Arquitecturas Paralelas (GAP) del Departamento de Informática de Sistemas y Computadores (DISCA) de la Universitat Politècnica de València (UPV). This architecture is included in the european project MANGO (`http://www.mango-project.eu`). The developed network with quality of service support will be used in that project to guarantee various bandwiths and latencies to different applications with temporal requirements.

The network implements virtual networks, each containing a set of virtual channels. Virtual networks enables to divide different traffic in a logic way and therefore is capable to set aside bandwith from the distinct virtual networks with the purposes of guaranteing quality of service.

The work involves the design of a multitude of tests and real runnings with whole architecture working on an own operating system that verifies and validates each component along with the different configurations of the interconnecion network. The synthesis of each of the components and the switch (in it's different configurations) has also been performed with the goal of obtaining the resource usage on an FPGA system. During the development of this project different commercial tools have been used such as Xilinx Vivado, version control software (Git) and other private ones such the PEAKos operating system.

Keywords: networks-of-chip, quality of service, FPGA, multicore systems, virtual channels, virtual networks, configurable.

Índice de Abreviaturas

ANSI (*American National Standards Institute*), Instituto nacional estadounidense de estándares.

BE (*Best Effort*).

BUS (*Bus*).

DISCA (*Department of Computer Engineering*), Departamento de Informática de Sistemas y Computadores.

DOR (*Dimmension Order Routing*), Encaminamiento por orden de dimensiones.

FPA (*Fixed Priority Arbiter*), Árbitro de prioridades fijas.

FPGA (*Field Programmable Gate Array*), Dispositivo de puertas lógicas programables.

GAP, Grupo de Arquitecturas Paralelas.

GS (*Guaranteed Services*), Servicio garantizado.

HOL (*Head Of Line blocking*), Bloqueo de encabezado de línea.

HPC (*High Performance Computing*), Computación de alto rendimiento.

IB (*Input Buffer*), Cola de entrada (también referenciada como IBUF).

IC (*Integrated Circuit*), Circuito integrado.

IP (*Intellectual Property*), Propiedad intelectual.

ISA (*Instruction Set Architecture*), Conjunto de instrucciones.

KB (*Kilobyte*), Kilobyte.

LBDR (*Logic-Based Distributed Routing*), Encaminamiento distribuido basado en lógica.

LOG (*Logic*), Lógica.

LUT (*Lookup Table*), Tabla de consulta.

L1 (*First Level Cache*), Memoria cache de primer nivel.

L1D (*First Level Data Cache*), Memoria cache de datos de primer nivel.

L1I (*First Level Instruction Cache*), Memoria cache de instrucciones de primer nivel.

L2 (*Second Level Cache*), Memoria cache de segundo nivel.

MANGO (*exploring Manycore Architectures for Next-GeneratiOn HPC systems*), Explorando arquitecturas para la nueva generación de los sistemas multinúcleo de alto rendimiento.

MIPS (*Microprocessor without Interlocked Pipeline Stages*), Microprocesador sin las etapas del *pipeline* entrelazadas.

MS (*Message System*), Generador de tráfico sintético.

MUX (*Multiplexer*), Multiplexor.

NI (*Network Interface*), Interfaz de red.

NID (*Network Of IDs*), Red de identificadores.

PC (*Personal Computer*), Computadora personal.

PEAK (*Partitioned-Enabled Architecture for Kilocores*), Arquitectura con particionado para kilocores.

POSIX (*Portable Operating System Interface of UNIX*), Interfaz de Sistema Operativo Portable de UNIX.

QoS (*Quality Of Service*), Calidad de servicio.

RT (*Routing*), Unidad de encaminamiento.

SA (*Switch Alocator*), Árbitro del puerto de salida.

SoC (*System On Chip*), Sistema en el chip.

TR (*Tile Register*), Banco de registros especial del tile.

RR (*Round Robin*), Árbitro de selección cíclico.

RTT (*Round Trip Time*), Tiempo de transmisión de ida y vuelta del enlace.

XOP (*Crossbar Output*), Unidad de salida del crossbar.

VA (*Virtual Alocator*), Árbitro de canales virtuales.

VC (*Virtual Channel*), Canal virtual.

VN (*Virtual Network*), Red virtual.

2D (*Two Dimension*), Dos dimensiones.

3D (*Three Dimension*), Tres dimensiones.

Índice General

Índice de Figuras

Índice de Tablas

CAPÍTULO 1

Introducción

Los sistemas multinúcleo son ya una realidad y ofrecen un rendimiento de cálculo superior para un mismo margen de consumo. El uso de decenas o centenares de procesadores sencillos (muy eficientes desde el punto de vista energético) está reemplazando a procesadores tradicionales mas veloces pero poco eficientes energéticamente. Esto nos lleva a sistemas con chips que incluyen ya centenares de procesadores, como es el caso de los chips ofrecidos por KALRAY (`http://www.kalrayinc.com`). En estos sistemas se hace imprescindible el uso de una red dentro del propio chip. La red, conocida en el mundo académico e industrial como *network-on-chip* (NoC) debe diseñarse de forma eficiente, para evitar que se convierta en el cuello de botella del sistema limitando sus prestaciones.

Las redes en el chip se están investigando durante los últimos 15 años [1]. Aunque se han tratado y propuesto numerosas técnicas, un aspecto importante en el diseño de las redes en el chip es la incorporación de técnicas que faciliten a las aplicaciones que se están ejecutando de diferentes garantías de calidad de servicio (QoS). Así, una red con QoS debe garantizar anchos de banda y latencias máximas de transmisión.

En el presente trabajo se ha dotado a una red en el chip existente de soporte de calidad de servicio. Inicialmente la red de datos del sistema se encuentra implementada por dos redes físicas que han sido reemplazadas por una sola red que implementa dos redes virtuales (Virtual Network, VN). Por otra parte, para poder estudiar en profundidad el comportamiento de esta nueva red, se ha parametrizado toda la implementación para así poder activar o desactivar los componentes que se necesiten en cualquier momento y así estudiar diferentes configuraciones. Estas configuraciones instancian un número de recursos diferentes y por lo tanto sus prestaciones varían.

La configuración más completa está compuesta por dos VNs de cuatro canales virtuales (Virtual Channel, VC) en cada red virtual junto con árbitros por pesos en los puertos de salida para así poder garantizar un ancho de banda determinado a cada red virtual. El hecho de disponer de redes virtuales nos permite dar so-

porte de calidad de servicio porque la lógica de las redes se encuentra totalmente separada y por lo tanto los paquetes de diferentes redes virtuales no interfieren entre ellos. Los canales virtuales proporcionan un desacoplamiento entre los canales físicos y las colas de almacenamiento a la entrada de cada conmutador por lo que ayudan a obtener una mayor utilización de los enlaces físicos y evita la contención de mensajes.

Otros aspectos importantes de la red diseñada es que garantiza la entrega en orden, evita la pérdida de paquetes y obtiene una utilización eficiente de los enlaces físicos entre conmutadores. La validación y la puesta a prueba de la red ha sido llevada a cabo en un sistema de placas FPGAs y ejecutándose bajo un sistema operativo dedicado.

1.1 Contexto y Motivación

El trabajo desarrollado está íntimamente ligado con el proyecto europeo MANGO (`http://www.mango-project.eu`) iniciado el mes de octubre del 2015. Este proyecto se encuentra liderado por el Grupo de Arquitecturas Paralelas (GAP) del Departamento de Informática de Sistemas y Computadores (DISCA) de la Universitat Politècnica de València. El objetivo del proyecto es la reducción al máximo del consumo energético de los grandes centros de cálculo computacional sin afectar ni a su rendimiento ni a las prestaciones de las aplicaciones en ejecución. En él se plantea el desarrollo de un nuevo prototipo de sistema masivo de cómputo de altas prestaciones que permitirá mejorar el rendimiento de los elementos de cómputo y la interconexión de sus componentes, gestionar de forma más eficiente los recursos y optimizar el control de energía y los modelos de programación y refrigeración.

En prototipo consta de sistemas heterogéneos de altas prestaciones compuestos por procesadores con grandes capacidades de cómputo, procesadores de bajo consumo, *GPUs*, aceleradores *hardware* y dispositivos de prototipado (*FPGAs*). La infraestructura de interconexión entre todos estos dispositivos necesita de una utilización eficiente de los recursos para cumplir con los requisitos de las aplicaciones que van a ser ejecutadas en este sistema de HPC (High-performance Computing). Algunas de las aplicaciones de HPC son tales como: transcodificación de video en tiempo real, aplicaciones de imágenes médicas, de seguridad, Big Data y otras aplicaciones de comunicación. Todas esas aplicaciones tienen requisitos temporales, lo que se traduce en necesidades de soporte de Calidad de Servicio (QoS) en la propia red.

Adicionalmente, en este proyecto se está utilizando la arquitectura PEAK como punto de partida. Esta arquitectura ha sido desarrollada íntegramente por el GAP. PEAK es una arquitectura de procesador multinúcleo con memoria compartida. El objetivo de PEAK es el uso de técnicas de emulación en sistemas FPGA para la investigación (así como para la docencia) en temas de diseño de nuevas

arquitecturas de procesador multinúcleo. PEAK permite múltiples variantes en el diseño de la arquitectura, partiendo de la definición de la red de interconexión, definiendo completamente los conmutadores, los protocolos de encaminamiento, la jerarquía de memoria, protocolos de coherencia entre memorias cache y, así mismo, define el modelo de programación y el protocolo de acceso al sistema desde un computador externo.

Por otro lado, una red en el chip nos permite tener una comunicación entre los distintos nodos dentro del mismo chip. Los mensajes que viajan por la red son encaminados por los conmutadores para llegar a su nodo destino. A lo largo de todo el proceso hay que tomar varias decisiones para que la red funcione correctamente y todos los mensajes lleguen a su destino. Algunos aspectos clave son las técnicas de conmutación, el control de flujo, el encaminamiento, la tolerancia a fallos y la calidad de servicio.

El conmutador de partida del presente proyecto no soporta canales virtuales, por tanto, tiene una sola cola en cada puerto de entrada donde se almacenan todos los mensajes. Esto hace que si un mensaje se bloquea temporalmente, éste generará contención en el acceso al puerto de entrada, por lo que se degradarán las prestaciones. Al disponer de canales virtuales, los conmutadores aliviarían este efecto ya que otros mensajes podrían utilizar el puerto de entrada al tener asignados otros canales virtuales distintos al que está siendo ocupado por el mensaje bloqueado. Por lo tanto, contribuirán a un mejor funcionamiento de la red, se alcanzarán mayores prestaciones y una mejor gestión de los recursos de la red. El uso de canales virtuales también está ligado al desarrollo de algoritmos de encaminamiento más sofisticados, entre ellos los algoritmos adaptativos. En este trabajo no se pretende desarrollar técnicas de encaminamiento adaptativas, ni implementar estrategias de calidad de servicio. La arquitectura PEAK no tiene soporte a calidad de servicio (QoS), por lo que no puede ejecutar de forma adecuada las aplicaciones que se van a utilizar en el proyecto MANGO.

Toda la infraestructura de red en PEAK necesita provisionarse de soporte para calidad de servicio. El objetivo principal de este trabajo consiste en el desarrollo e implementación de soporte de calidad de servicio en la red de interconexión de la arquitectura PEAK, la cual será utilizada en el proyecto europeo MANGO.

1.2 Objetivos

A continuación se muestran los objetivos parciales que se fijaron inicialmente para el proyecto, con el fin de alcanzar el objetivo global.

- El objetivo del trabajo consiste en rediseñar e implementar las unidades necesarias para proveer calidad de servicio (QoS) a la red de la arquitectura PEAK.

- Diseñar distintas versiones de árbitros de canales virtuales y árbitros de puertos. Estos árbitros permitirán la gestión de distintas estrategias de QoS.
- Añadir a PEAK el concepto de redes virtuales (Virtual Networks, VN) que separan distintos flujos de mensajes.
- Realizar tests de prueba para validar el correcto funcionamiento de las implementaciones desarrolladas.
- Añadir el soporte de redes virtuales y canales virtuales a la interfaz de red de la arquitectura PEAK.
- Realizar tests para verificar la unión de los inyectores con el conmutador.
- Elaborar y efectuar pruebas con una red 2×2 de cuatro conmutadores.
- Modificar el cableado de toda la arquitectura para reemplazar las dos redes de datos existentes por una sola red de datos que implementa QoS.
- Efectuar la simulación del sistema real.
- Efectuar la simulación de un sistema compuesto por ocho procesadores formando una configuración 4×2.
- Implementar una primera versión del sistema en FPGA compuesto por dos procesadores para verificar su correcto funcionamiento.
- Implementar un sistema 4×2 en FPGA y verificar su correcto funcionamiento mediante la ejecución de una aplicación de multiplicación de matrices.
- Añadir la función de estadísticas de las redes al sistema operativo PEAKos.
- Diseño y validación de un inyector de tráfico sintético.
- Añadir la función al sistema operativo PEAKos para el uso del inyector de tráfico sintético.
- Comprobar el correcto funcionamiento del sistema compuesto por 8 procesadores funcionando con varios flujos de trafico inyectados a una tasa determinada y verificar las reservas de ancho de banda para las redes virtuales.

Nótese que el objetivo principal del presente trabajo es el soporte de los mecanismos básicos a nivel de red para proveer QoS al sistema. El objetivo no es la utilización eficiente y efectiva de dicho mecanismo. Tal objetivo será alcanzado en metas posteriores a este trabajo y en el marco del proyecto europeo MANGO.

El presente trabajo se estructura de la siguiente forma. En el Capítulo 2 se presentan conceptos previos y una breve descripción del estado del arte en técnicas

de QoS. En el Capitulo 3 se describe con detalle la arquitectura PEAK, base de partida de nuestro trabajo. En el Capítulo 4 se describen todos los desarrollos realizados, confiriendo a PEAK con soporte a QoS. En el Capítulo 5 se realizan los diferentes experimentos de evaluación y test. Finalmente, el Capítulo 6 refleja conclusiones y trabajo futuro.

Adicionalmente se han realizado otras tareas que no se ven reflejadas en esta memoria:

- Implementación de contadores para las estadísticas de la red.
- Implementación de la función *suffle* y *unsuffle* en algunos árbitros *round-robin*. Esta función consiste en el barajado de los bits que contienen los vectores de peticiones. Se realiza un barajado en los vectores con peticiones provenientes de varios puertos de entrada o de diferentes redes virtuales dependiendo del caso. Esta función tiene como objetivo recolocar los bits de tal forma que el árbitro *round-robin* gestione las peticiones de la forma mas equitativa posible entre canales virtuales de la misma red virtual y entre redes virtuales.
- Cableado entre todos los módulos a lo largo de todas las capas de PEAK.
- Adición de comandos al sistema operativo PEAKos. El comando *nshow* muestra las estadísticas de la red, *reset_stats* pone a cero las estadísticas del sistema y *switch_inyect* modifica un registro en uno de los *tiles* que activa el inyector de tráfico sintético.

CAPÍTULO 2

Conceptos Previos y Estado del Arte

En este capítulo se presenta un breve resumen de los conceptos básicos relacionados con el trabajo desarrollado, así como una revisión del estado del arte en la temática del trabajo.

2.1 Red en el Chip y Sistema en Chip

Una red en el chip (NoC; Network-On-Chip) es, como su nombre indica, un subsistema de comunicación en un circuito integrado (IC), el cual conecta nodos lógicos conocidos como IP *cores* dentro de un sistema en chip (SoC; System-On-Chip).

La tecnología NoC usa técnicas, métodos y algoritmos derivados de las redes de interconexión para sistemas de cómputo. Ahora bien, en el campo de las NoC, estos métodos deben ser ajustados a las limitaciones existentes en el propio chip, las cuales son diferentes de las redes de interconexión tradicionales.

Las redes NoC tienen por objetivo proveer de las prestaciones necesarias en la comunicación para el propio chip, siendo eficientes no solo desde el punto de vista de prestaciones sino también desde el punto de vista energético.

En las siguientes secciones se introducen los conceptos típicos de las redes de interconexión, particularizando para las redes NoC. Después se ofrece un estudio del arte relacionado con las redes NoC y la calidad de servicio.

2.2 Topologías de Interconexión

El primer paso en el diseño de una red consiste en determinar una topología de interconexión. La topología define el patrón de interconexión de los distintos dispositivos de una red. La topología tiene un gran impacto en el coste y las prestaciones del sistema final. Este impacto se debe por ejemplo al número de enlaces y conmutadores, diámetro de la red y número de comunicaciones paralelas posibles (determinado por la bisección). Otros impactos directamente relacionados con la implementación física son el retardo de propagación de la señal, ciclo de reloj de la red, área consumida, energía consumida, etc.

Actualmente podemos diferenciar cuatro categorías de topologías principalmente:

- **Redes de medio compartido:** todos los nodos del sistema comparten un mismo medio de comunicación como por ejemplo un bus.
- **Redes conmutadas:** La red está formada por conmutadores y los nodos no comparten todos los conmutadores, existiendo la posibilidad de comunicación concurrente. Podemos distinguir:
  - **Redes directas:** Los nodos incorporan un conmutador y estos se interconectan por medio de enlaces punto a punto.
  - **Redes indirectas:** Los nodos se conectan por medio de conmutadores y los conmutadores se interconectan con enlaces punto a punto.
  - **Redes híbridas**, incluyendo combinaciones de las anteriores.

Las redes directas son redes punto a punto con una topología regular puesto que cada nodo dispone de un conmutador. Por otro lado, las redes indirectas efectúan la comunicación punto a punto a través de conmutadores que pueden conectarse con ninguno, uno o varios nodos y con otros conmutadores entre si. Estas redes suelen tener forma irregular multietapa. En la Figura 2.1 se muestra un ejemplo de redes directas e indirectas. En este trabajo se implementa una red directa, en concreto una malla 2D.

Las redes en el chip deben ocupar muy poca área y tener un consumo energético reducido. Cabe destacar que se implementan en un único plano y es por ello que durante muchos años las redes en 2D han sido muy comunes en el chip. De hecho, la malla 2D es muy común puesto que su mapeado en un plano bidimensional es muy intuitivo, la longitud de los enlaces es regular y permite un alto nivel de modularidad. El área crece linealmente con el número de nodos. Por lo contrario, el ancho de banda de la bisección crece linealmente al aumentar cuadráticamente el tamaño del sistema, esto supone un problema de escalabilidad del sistema.

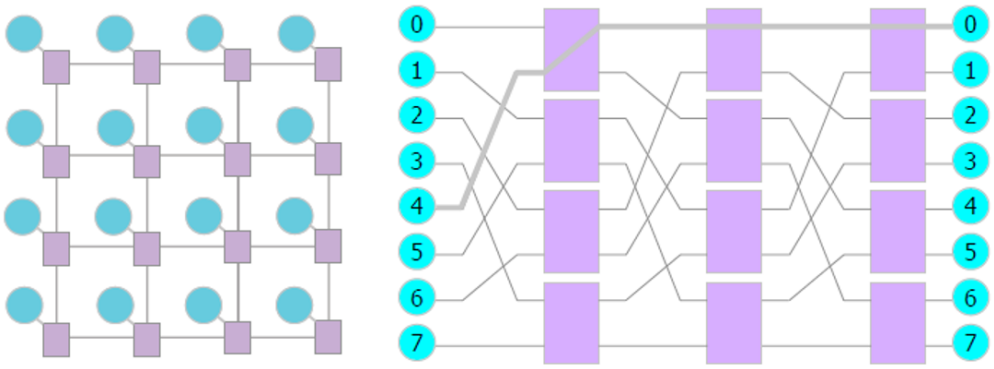

Figura 2.1: Ejemplo de topologías directas e indirectas. Nodos representados como círculos y conmutadores como cuadrados

2.3 Técnicas de Control de Flujo

Las técnicas de control de flujo tienen como principal objetivo evitar la pérdida de información y por lo tanto sincronizar la transferencia de unidades de información entre un transmisor y un receptor. Como consecuencia, se establece un diálogo (protocolo de comunicación) entre dos nodos adyacentes.

La sincronización de las comunicaciones se realiza sobre unidades de información o de control de flujo. Estas unidades de información representan el tamaño mínimo de información en el que puede descomponerse y se denomina *flit*.

Un mensaje puede descomponerse en paquetes, los cuales incorporan información de encaminamiento (por lo tanto cada paquete es una unidad de transferencia independiente). A su vez, un mensaje o un paquete es descompuesto en *flits*. En este caso, existe un *flit* de cabecera con información de encaminamiento y *flits* de *payload*. También se identifica un *flit* de cola (*tail*).

El tamaño del *flit* es variable y depende de las diferentes implementaciones de redes NoC. De hecho, un *flit* puede ser del tamaño de un mensaje (o paquete) completo o de unos cuantos bits del mismo. El envío de un *flit* nunca se realiza hasta tener la certeza de que su almacenamiento se producirá sin pérdida de éste en el receptor.

La transferencia de un *flit* se puede descomponer a nivel físico en *phits* representando la unidad de información que puede ser transferida en un solo ciclo de reloj por el canal. Normalmente, en las redes en el chip, el tamaño de un *flit* equivale al tamaño de un *phit*.

Existen dos técnicas predominantes para implementar el control de flujo en redes de interconexión: Stop&Go y créditos. A continuación describimos cada una de estas técnicas.

2.3.1. Control de Flujo Stop & Go

La técnica Stop & Go utiliza dos umbrales en la cola de recepción para notificar a la fuente de la posibilidad (o no) de transmitir *flits*. De hecho, actúa como un semáforo en el transmisor que se activa según dos niveles (superior e inferior). Tal y como se muestra en la Figura 2.2 en la cola se sitúan dos umbrales de *STOP* y *GO*.

Una vez la ocupación de la cola supera el umbral *STOP*, el receptor envía una señal de control al emisor indicándole que pare la transmisión de *flits* (se pone el semáforo en rojo). Puesto que no se inyectan *flits*, la cola se vacía progresivamente (en la medida que los *flits* son procesados por el receptor) hasta que la ocupación de la cola en *flits* supera por abajo el umbral *GO*, instante en el cual el receptor envía una señal de reactivación de la transmisión de *flits* (semáforo en verde).

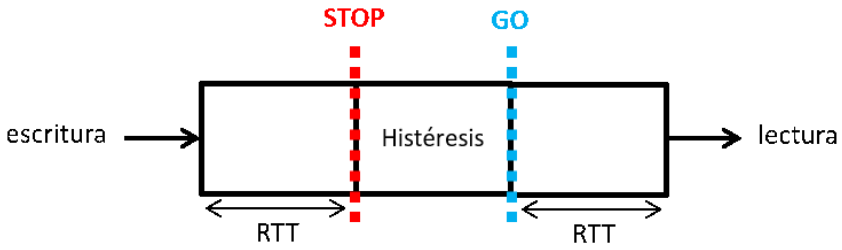

Figura 2.2: Tamaño de cola con Stop & Go

El tamaño de las colas debe seleccionarse de tal manera que se aproveche el 100 % del ancho de banda del enlace. En concreto, la capacidad de almacenamiento entre el umbral *STOP* y la cima de la cola así como la capacidad de almacenamiento entre el umbral *GO* y la base de la cola debe ser igual o mayor al tiempo de transmisión de ida y vuelta del enlace (*round-trip time*). Cabe destacar que es preferible dejar un tamaño de histéresis en la cola para que no se activen y desactiven las señales de notificación continuamente.

2.3.2. Control de Flujo por Créditos

Como su propio nombre indica, esta técnica de control de flujo dispone de un contador con el número de créditos que puede disponer en un momento dado. Típicamente un crédito se corresponde con una unidad de transmisión de información, pudiendo ser desde un *flit*, un conjunto de *flits*, un paquete completo o incluso un mensaje. Asumimos para la descripción del mecanismo que la unidad de crédito es el *flit*. Por tanto, cuando se transmite un *flit* el contador se decrementa en una unidad. El receptor, cuando libera un crédito, lo envía de vuelta, a lo que el transmisor lo añade al contador. El transmisor puede enviar *flits* siempre que el contador no esté a cero.

La desventaja de esta técnica es el hecho de necesitar notificaciones periódicas al transmisor notificando nuevos créditos disponibles, con el consiguiente consumo de ancho de banda. Por el contrario, como se muestra en la Figura 2.3 el tamaño de las colas se reduce en este caso al *round-trip time*, siendo menor el tamaño en colas que en la técnica anterior de Stop & Go.

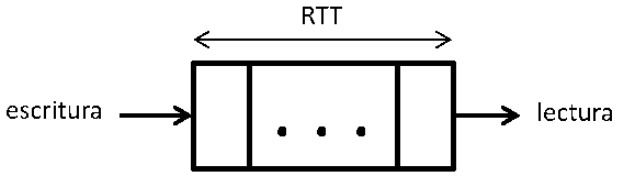

Figura 2.3: Tamaño de cola con control de créditos

2.4 Técnicas de Conmutación

Las técnicas de conmutación implementan cuando y cómo avanzan los mensajes a través de los conmutadores, intentando minimizar el tiempo de asignación de los recursos a los mensajes con el fin de maximizar las prestaciones. Cabe destacar que también determina cómo se gestionan los mensajes en el caso de que queden detenidos en la red. A continuación se explican algunas de las técnicas de conmutación mas conocidas para las redes de interconexión y que también se aplican en las redes NoC.

2.4.1. Conmutación Virtual Cut-Through

En la conmutación *virtual-cut-through* se transmiten paquetes, y una vez se recibe la cabecera del paquete se inicia el encaminamiento y su retransmisión al siguiente conmutador, antes de que el paquete haya llegado por completo al conmutador. En el caso de que el paquete no pueda avanzar (por ejemplo, por estar la salida del conmutador ocupada), el paquete se debe almacenar por completo en el conmutador (en la cola asignada del puerto de entrada).

Por tanto, una de las principales características de esta técnica de conmutación es que la cola de cada nodo debe ser lo suficientemente grande como para almacenar un paquete completo. Por tanto, antes de permitir el avance del paquete se debe garantizar la existencia de suficiente espacio de almacenamiento (créditos) en el siguiente conmutador. En este método de conmutación la unidad de control de flujo es el paquete. Nótese, por tanto, la necesidad de paquetizar mensajes en la fuente, con el consiguiente sobrecoste.

2.4.2. Conmutación Wormhole

En la conmutación *wormhole* el mensaje se descompone en *flits* y sólo el *flit* de cabecera posee información para el encaminamiento. La unidad de control de flujo es el *flit*.

Al igual que en la técnica anterior, el avance de los *flits* se puede producir una vez se recibe el *flit* de cabecera. Sin embargo, cuando el mensaje no puede avanzar, el mensaje permanece en la red ocupando los recursos (canales, colas) a lo largo de toda la ruta utilizada hasta ese momento. De esta forma no es necesario que las colas de cada nodo almacenen íntegramente los mensajes por completo, por lo que el tamaño de las colas es menor que en la técnica anterior. De hecho, las colas deben tener el tamaño mínimo para cumplir el *round- trip time* del enlace.

La conmutación *wormhole* nos lleva a una gestión más eficiente del espacio de almacenamiento y, como consecuencia, un menor consumo de energía en la NoC. Ahora bien, *wormhole* puede conllevar una mayor congestión en la red debido a que un mensaje puede quedar temporalmente bloqueado, manteniendo ocupados varios recursos de la red. En la Figura 2.4 podemos comparar el comportamiento de las dos técnicas de conmutación.

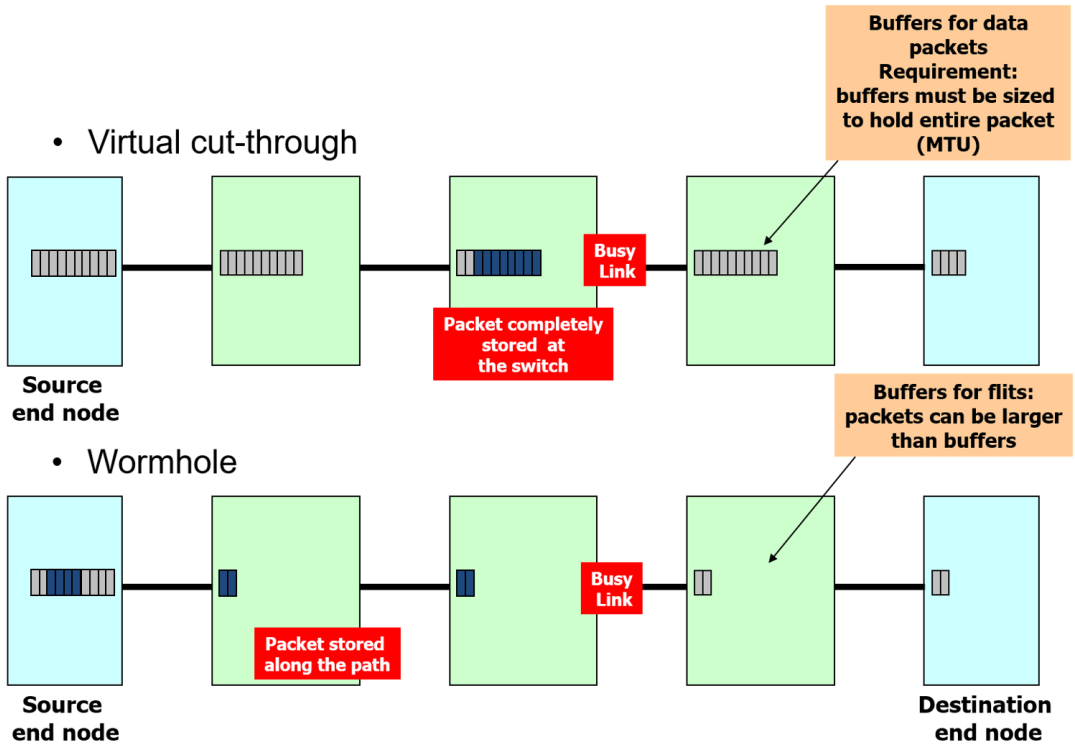

Figura 2.4: Comparación entre *virtual-cut-through* y *wormhole*

2.5 Canales Virtuales y Redes Virtuales

Como hemos comentado anteriormente, la técnica de conmutación *wormhole* nos lleva a mayores índices de congestión en la red. Para aliviar este hecho se utilizan canales virtuales (VCs; *virtual channels*). El uso de VCs es compatible también con la conmutación *virtual cut-through*.

Cada canal físico se descompone en varios canales lógicos o virtuales multiplexándose en el tiempo. Cada canal virtual tiene asociado una cola particular, y a cada una de ellas se le aplica un control de flujo individual (por ejemplo, cada canal virtual tiene sus propios umbrales de *stop* y *go* o créditos). La Figura 2.5 muestra la organización en canales virtuales. El hecho de disponer de varias colas de entrada nos proporciona un desacoplamiento entre los canales físicos y las colas de almacenamiento.

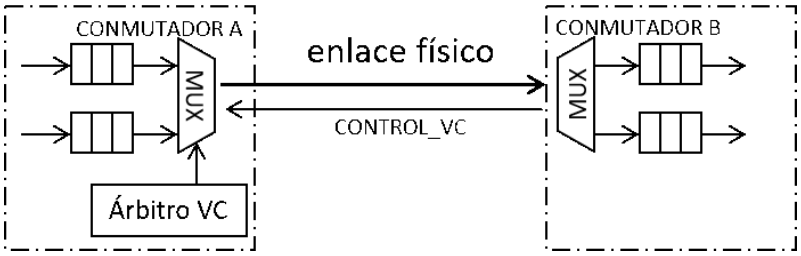

Figura 2.5: Canales virtuales en cada puerto del conmutador

Al usar canales virtuales, un mensaje bloqueado no mantiene ocupado el canal físico sino solamente el canal virtual que tenga asignado. De esta forma, otros mensajes pueden avanzar por el canal físico al utilizar otros canales virtuales. Como consecuencia, un mensaje que no pueda avanzar a través del conmutador no impide que otro lo adelante, con el consiguiente aumento de la eficiencia de la red.Véase la Figura 2.6.

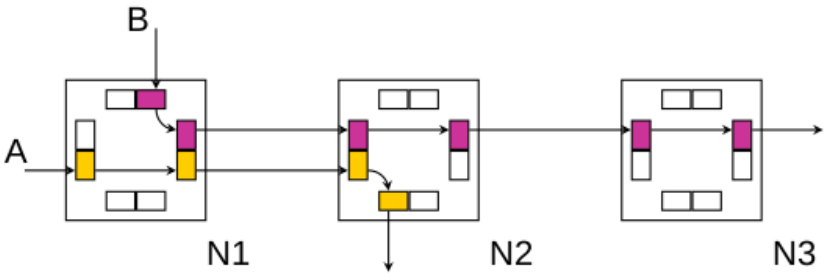

Figura 2.6: Ejemplo de funcionamiento de los canales virtuales

Los canales virtuales originalmente se utilizaban como solución al problema de la elevada congestión en la red [2, 3]. Sin embargo, también se pueden utilizar para otros fines, por ejemplo incrementar las prestaciones, la evitación de bloqueos por encaminamiento, y la provisión de calidad de servicio.

El concepto de redes virtuales (VNs; *virtual networks*) aparece cuando se ofrece calidad de servicio al separar de forma lógica los recursos por los que se transmiten mensajes con diferentes niveles de prioridad. Estas redes virtuales utilizan recursos separados (colas) y los mensajes en una red virtual no utilizan recursos de otras redes virtuales. Además, dentro de cada red virtual podemos tener diferentes canales virtuales implementados. La Figura 2.7 muestra un ejemplo donde tenemos dos redes virtuales con dos canales virtuales por red virtual. En este trabajo nos basamos en el uso de canales virtuales y redes virtuales para proveer soporte de calidad de servicio.

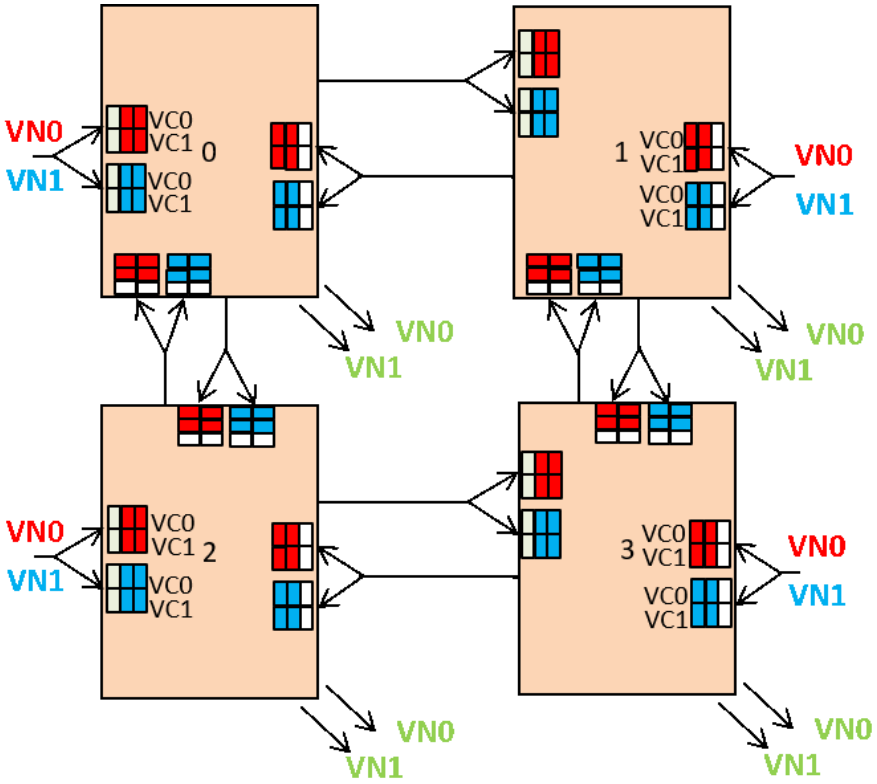

Figura 2.7: Ejemplo de una red 2x2 (malla 2D) con 2 VNs y 2VCs por red virtual

2.6 Algoritmos de Encaminamiento

Un aspecto fundamental en las redes es el encaminamiento, el cual nos indica por donde deben encaminarse los mensajes o paquetes. Por tanto, el algoritmo de encaminamiento nos define la ruta que ha de seguir al mensaje desde un nodo origen a través de los conmutadores de la red. El mensaje o paquete hará uso de los recursos de la red según las técnicas de conmutación y control de flujo implementadas. El principal objetivo de un buen algoritmo de encaminamiento consiste en distribuir los paquetes a lo largo de toda la red de forma equilibrada,

minimizando la contención de la red y haciendo uso del mayor número de rutas posibles.

Los algoritmos pueden ser deterministas o adaptativos. En el encaminamiento determinista un mensaje sigue siempre la misma ruta entre un origen y un destino, independientemente del estado de la red. Por contra, en el encaminamiento adaptativo, el mensaje puede tomar diferentes alternativas a lo largo de su ruta, principalmente en función del estado de la red en ese momento.

El encaminamiento determinista tiende a ser simple y, por lo tanto, la implementación *hardware* resulta mas eficiente y compacta. También garantiza la entrega en orden de los mensajes ya que la ruta depende únicamente del nodo en el que se encuentra y el destino. Por otra parte, el encaminamiento adaptativo no garantiza la entrega en orden pero ofrece rutas alternativas e incrementa la flexibilidad del encaminamiento. Todo ello nos lleva a una implementación con mayor complejidad.

Uno de los aspectos a tener en cuenta en el encaminamiento es la posibilidad de situaciones de bloqueo donde los mensajes se bloquean entre ellos y nunca llegan a su destino. Podemos encontrar tres situaciones de bloqueo: *deadlock*, *livelock* y *starvation*. La mas común es la primera que se produce cuando los mensajes no pueden avanzar debido a que los recursos (colas) que solicitan se encuentran ocupados. Estos mensajes pueden estar bloqueando a otros y así consecutivamente hasta formar un ciclo entre ellos con lo que nunca podrán avanzar, tal y como se muestra en la Figura 2.8.

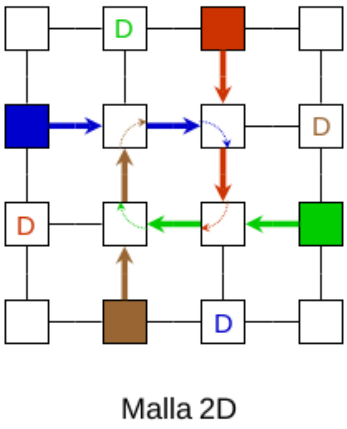

Figura 2.8: Situación de bloqueo entre mensajes

Un algoritmo de encaminamiento es libre de esta situación (*deadlock*) cuando su grafo de dependencia de canales es acíclico. El grafo de dependencia de canales representa todas las dependencias entre dos canales que introduce el algoritmo de encaminamiento en cuestión. Existe dependencia entre dos canales c_i y c_j si el encaminamiento solicita el canal c_j para un mensaje que está ocupando el canal c_i. Al construir el grafo, y al no encontrar ciclos en él, podemos asegurar que el algoritmo de encaminamiento es libre de la situación de *deadlocks* a nivel

de red. Existen otros algoritmos que permiten la formación de ciclos en su grafo y aún así son libres de bloqueo (para mayor detalle, se recomienda leer la teoría de Duato [2]).

Los problemas de *livelock* vienen cuando un algoritmo de encaminamiento utiliza rutas no mínimas y un mensaje es reencaminado siempre por la red no llegando nunca a su destino, posiblemente porque tenga menor prioridad que otros mensajes. Este problema se resuelve con un mecanismo de arbitraje que evite esta situación de forma indefinida a un mensaje. De la misma forma, el problema de la inanición (*starvation*) ocurre cuando un mensaje no es encaminado satisfactoriamente nunca, posiblemente por tener menor prioridad en el acceso a recursos. De nuevo, un arbitraje equitativo para todos los mensajes evita esta situación.

En este trabajo utilizamos el algoritmo de encaminamiento DOR (dimension order routing), el cual es determinista y tiene un grafo acíclico. El algoritmo DOR limita el encaminamiento a hacer un recorrido ordenado de los recursos por dimensiones de la red y si lo adaptamos a una red de dos dimensiones como es nuestro caso (malla 2D) obtenemos el encaminamiento XY. La Figura 2.9 muestra un ejemplo del algoritmo XY, resolviendo el bloqueo mostrado en la figura anterior.

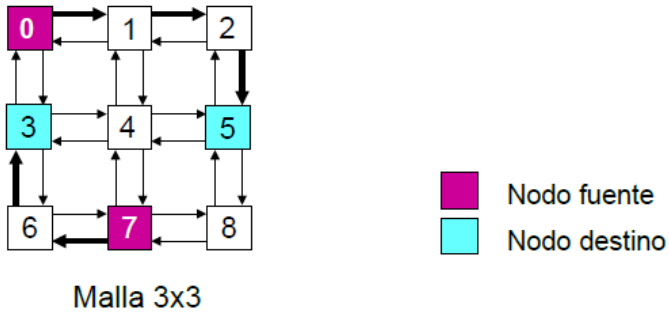

Figura 2.9: Algoritmo de encaminamiento XY en una malla 3x3

2.7 Arbitraje

La unidad de arbitraje se encuentra en cada conmutador de la red. La unidad de arbitraje actúa cuando dos o mas mensajes hacen peticiones al mismo puerto de salida en un mismo instante de tiempo. En ella se determina qué mensaje obtiene acceso al recurso. Esta función puede implementarse de forma centralizada o distribuida. Si se implementa de forma centralizada, el árbitro tendrá como señales de entrada las peticiones y las señales de estado de todo el conmutador. Por lo contrario, si es de forma distribuida los árbitros se verán repartidos entre los distintos puertos de entrada y/o salida.

Un buen árbitro debe maximizar la asignación de recursos a las peticiones de los mensajes pero existen algunos problemas como la inanición. Este problema surge cuando los mensajes nunca obtienen acceso al recurso. La solución a la inanición es que los árbitros sean equitativos cuando conceden permisos a los recursos compartidos y es por ello que normalmente se implementan árbitros *round-robin*. Existen algunas técnicas de arbitraje distribuido para conmutadores: árbitros de dos fases, de tres fases e iterativos.

2.8 Estado del Arte

En el área de las redes NoC existe gran cantidad de trabajos de investigación en muy diversos temas, desde topologías, algoritmos de encaminamiento, mecanismos de control de flujo, optimización de recursos, hasta nuevas tecnologías (y sus implicaciones) como son las redes fotónicas y las redes inalámbricas dentro del chip.

También existe gran variedad de trabajos relacionados con la calidad de servicio (QoS; *Quality-Of-Service*). Cuando se habla de calidad de servicio se hace referencia a todo lo relacionado con garantizar tiempos de entrega de los mensajes (latencias), garantizar ancho de banda para unas determinadas aplicaciones, redundando en la garantía de ofrecer una determinada productividad en el sistema, priorizando un tráfico sobre otro. Normalmente, los trabajos asumen la existencia de un tráfico con necesidad de garantía de servicio (GS; *Guaranteed Services*) con requisitos temporales de entrega de mensajes, frente a otro tipo de tráfico de aplicaciones de uso general (BE; *Best Effort*) que no tienen necesidad de ningún requisito temporal.

En [6] se trabaja una solución de compromiso en el diseño de un conmutador con los dos tipos de tráfico (GS y BE). En concreto, se contrasta dos arquitecturas del conmutador, una para cada tipo de datos y llega a una arquitectura que ofrece un compromiso (*trade-off*) entre complejidad *hardware* y eficiencia en la utilización de recursos. La arquitectura se basa en que el tráfico GS no se ve afectado en ningún momento por el trafico BE puesto que el tráfico GS usa los recursos que necesita y el BE los que encuentra libres. De esta forma, se garantiza la integridad de datos, sin pérdidas en los envíos y entrega en orden. Además, el tráfico GS obtiene productividad y latencia garantizadas.

En [9] se provee de QoS, garantizando ancho de banda y latencia en una arquitectura de NoC denominada *Nostrum*. Los paquetes con ancho de banda garantizado (GS) utilizan circuitos virtuales implementados con una combinación de dos conceptos: contenedores en bucle y redes disjuntas temporales. Los contenedores garantizan el acceso a la red, sea cual sea su carga y sin pérdida de mensajes. Las redes disjuntas temporales se usan con el fin de llegar a algunos circuitos virtuales y también son usadas por el tráfico BE en la red.

Otra forma de garantizar QoS es la que se implementa en [10] utilizando una especie de cache central en el conmutador dividida para varios canales. Según una clasificación del tráfico por clases se usan árbitros que conceden prioridades en cada canal al tráfico existente clasificado como el más prioritario.

Así mismo, en [4] se presenta el concepto de tramas globalmente sincronizadas para una NoC que garantiza la QoS de la red referente al ancho de banda usado y una latencia acotada. El sistema solo tiene en cuenta unas pocas tramas de QoS (para reducir el coste de gestión) y estas tramas llevan información referente al retardo deseado en llegar al destino que se actualiza en cada punto de la red mediante una ecuación. De esta forma, en cada nodo de la red, el paquete marcado con menos retardo deseado que el de la trama de QoS será encaminado con la mayor prioridad y en este caso, se detiene la inyección de nuevos paquetes provenientes a esta trama. El mecanismo funciona como una barrera que separa grupos de paquetes entre sí.

Existen trabajos mas complejos como [11] en el que se combinan características con un enfoque basado en tramas de paquetes y otras basadas en la tasa de transferencia. Este sistema está provisto de QoS, fuerza la separación de flujos de datos y tiene una utilización eficiente del ancho de banda. No requiere de colas por cada flujo de datos y por lo tanto reduce el área y el consumo energético.

Algunos trabajos como [7] se centran en planificadores que mejoran la utilización de recursos. En concreto, se utiliza este planificador con el fin de optimizar la reserva de recursos en aplicaciones GS y de este modo disponer de mas recursos para aplicaciones BE. Junto con el planificador, también propone una nueva arquitectura del conmutador que mejora el rendimiento del sistema usando colas compartidas.

La calidad de servicio (QoS) se puede abordar también a través de un enfoque diferente. En [5] se usan herramientas de simulación para diseñar un conmutador con unas características específicas que cumpla con los requisitos temporales de las aplicaciones que se van a ejecutar. Estos requisitos pueden ser tales como el cumplimiento de unos tiempos de entrega de las tareas. La simulación tiene en cuenta la localización de los procesadores (núcleos) en la NoC debido a problemas relacionados con comunicaciones y prioridades. También combina junto con lo anterior, un mecanismo de control de flujo en las colas de entrada de forma que se incrementan las prioridades de los mensajes bloqueados y se desechan los viejos para mejorar el número de mensajes que cumplen sus tiempos de entrega en su destino.

Por otro lado, investigar un buen flujo de diseño para una NoC puede ser interesante. En [8] se presenta un flujo de trabajo para acelerar el diseño y la verificación de una NoC con un rendimiento determinado garantizado. Este flujo de diseño sirve para generar y configurar las instancias de una NoC que cumplan con los requisitos de aplicaciones específicas conociendo de antemano los requisitos de comunicación en la aplicación.

Por último, hay trabajos que apuestan por arquitecturas heterogéneas de NoCs que sean altamente escalables y con soporte de QoS. En la arquitectura propuesta en [12] el soporte de calidad de servicio se implementa solo en una o varias regiones. Esto es debido a que este soporte conlleva a utilizar más área y consumo de energía, por lo que solamente algunos nodos de la red se verán sobrecargados con estos recursos.

Como se ha visto en esta sección, existe una gran variedad de formas de afrontar la calidad de servicio para NoCs y este trabajo se ve enfocado en garantizar un ancho de banda determinado para diferentes flujos de mensajes. Se garantiza la integridad de datos, sin pérdidas en los envíos y entrega en orden. Además, la solución aportada en este trabajo está particularizada para la arquitectura PEAK y todo su ecosistema de gestión. En el siguiente capítulo se describe la arquitectura PEAK, punto de partida de este trabajo.

CAPÍTULO 3

La Arquitectura PEAK

PEAK es el acrónimo de la arquitectura base del proyecto. PEAK significa Partitioned-Enabled Architecture for Kilocores y ha sido desarrollado por el grupo de Arquitecturas Paralelas del departamento DISCA de la Universitat Politècnica de València. PEAK pretende desarrollar toda una arquitectura de un sistema multinúcleo totalmente operativa y que sirva tanto para fines docentes como de investigación. PEAK se utiliza en algunos proyectos de investigación tales como el proyecto europeo MANGO, también en proyectos académicos y su objetivo es estudiar nuevas arquitecturas multinúcleo para sistemas de cómputo masivo.

PEAK incluye un sistema operativo básico llamado *PEAKos* con gestión de interrupciones y excepciones, planificador de procesos e hilos y emulación de sistema de ficheros. También existe una *toolchain* de compilación con aplicaciones desarrolladas en *ANSI C* y *POSIX*. Todo ello nos permite desarrollar y depurar el sistema en un PC convencional.

Las diferentes instancias de la arquitectura PEAK se pueden ejecutar en un sistema de prototipado basado en múltiples FPGAs. Estas placas se conectan entre si formando sistemas de prototipado mas grandes, tal y como se muestra en la Figura 3.1. PEAK actualmente permite la emulación de un sistema de hasta 256 núcleos de procesamiento.

La arquitectura se encuentra en ciclo de desarrollo y utiliza el lenguaje de descripción hardware Verilog. Para ello se utilizan herramientas de simulación con lo que se comprueba el correcto funcionamiento de sus componentes. También se utiliza ficheros de programación de las FPGAs (*bitstreams*) con la implementación de la arquitectura, los cuales, una vez programadas las FPGAs, permiten utilizar la arquitectura PEAK en tiempo real.

La arquitectura PEAK está construida a partir de una estrategia de definición de parámetros de configuración, permitiendo así la generación de múltiples instancias con diferentes números de procesadores, redes, memorias, etc... con tan solo modificar algunos de estos parámetros. Esta flexibilidad en la definición de los parámetros de PEAK es muy importante, ya que permite de forma flexible el

Figura 3.1: Placas de prototipado multi-FPGA, incorporando cuatro dispositivos FPGA V2000T de Virtex, interconectados entre sí.

estudio de diferentes configuraciones, tanto para investigación como para docencia. A continuación se describen los diferentes componentes de PEAK así como su arquitectura general y detalles.

3.1 Arquitectura General de PEAK

PEAK está formado por una organización homogénea de componentes replicados a lo largo del sistema. Estos componentes se denominan *tiles* y engloban todos los módulos de PEAK. La Figura 3.2 muestra la organización de PEAK en *tiles* y como estos están conectados entre sí.

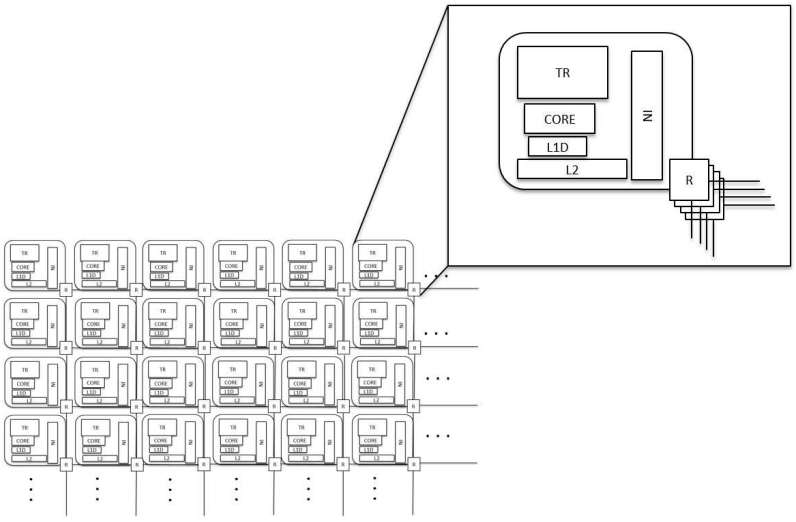

Figura 3.2: Estructura de la arquitectura en *tiles*

3.1.1. Componente Básico: el *Tile*

Todos los *tiles* en PEAK son idénticos y están formados por los siguientes componentes. En primer lugar tenemos uno o varios procesadores sencillos (denominados *core*) que implementan el ISA (instruction set architecture) de MIPS32. Estos procesadores son el elemento básico de cómputo de PEAK. Actualmente hay dos implementaciones desarrolladas. Una con ejecución en orden por medio de cinco etapas, con soporte de interrupciones y excepciones. Otra con ejecución fuera de orden mediante el algoritmo de *Tomasulo* y con soporte de coma flotante.[1] Cada procesador dentro del *tile* tiene asociado una memoria cache de instrucciones (incluida en el procesador) así como una cache de datos privada. Esta memoria cache (L1D) se puede configurar en PEAK tanto en tamaño como en comportamiento (protocolo de coherencia, descrito posteriormente). Véase la Figura 3.3.

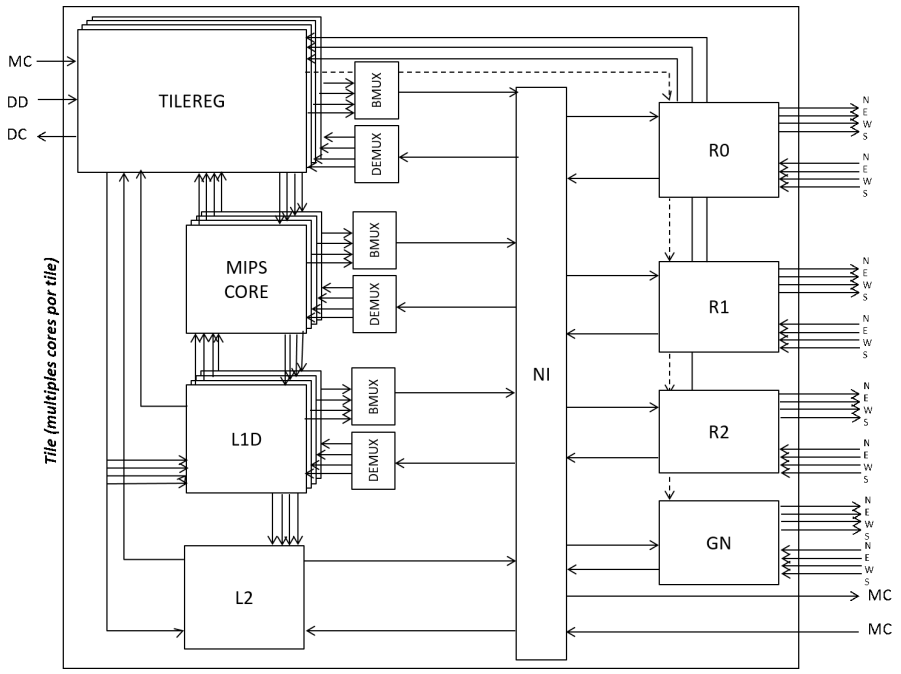

Figura 3.3: Estructura interna del *tile* con múltiples núcleos

Cada procesador está conectado también a un banco de registros especial, denominado TILEREG, que permite la configurabilidad y control de todos los elementos del sistema. En concreto, el TILEREG asociado al procesador puede detener y reanudar el procesador, cambiarle la dirección de ejecución, recolectar estadísticas del procesador, etc.

En el *tile* también se encuentra un banco de memoria de cache de segundo nivel (L2), el cual tiene una organización basada en memoria compartida pero

[1] Esta última versión del procesador fue desarrollada en un trabajo de fin de grado formado por cuatro alumnos donde uno de ellos es el autor de este trabajo.

distribuida. Esto quiere decir que el sistema completo de PEAK implementa una memoria cache de segundo nivel (L2) que tiene una capacidad igual a la suma de capacidades de todos los bancos de memoria L2 que se encuentran en todos los *tiles*. A su vez, esta memoria está compartida por todos los procesadores. Tanto la L1D como el banco L2 forman el sistema de memoria de PEAK (junto al controlador de memoria que se conecta al *tile*).

Todos estos componentes (procesadores, L1Ds, L2s, TILEREGs) se interconectan a través del interfaz de red (NI; *network interface*). Este módulo permite conectar los elementos entre ellos, implementando para ello colas de almacenamiento y un control de flujo. Podemos ver este módulo como un *crossbar* con colas. En PEAK también se implementan *tiles* de mas de un procesador y se replican tanto el número de caches L1, módulos TILEREGs y procesadores, así como las interconexiones entre sí y al NI.

Por último, en el *tile* también se encuentran los conmutadores que conectan los *tiles* entre sí. En PEAK se utilizan cuatro redes paralelas: VN0, VN1, VN2, NIDs. Las dos primeras redes (VN0 y VN1) se utilizan para transportar los mensajes de datos entre memorias cache (L1Ds y L2s). La tercera red (VN2) se utiliza para conectar los TILEREGs con los procesadores y con el exterior, así como las memorias cache L2s con el controlador de memoria. Por tanto, esta tercera red es una red global que permite configurar el sistema y acceder a memoria principal. Finalmente, la cuarta red (NIDs, *network of IDs*) es una red especial optimizada para envío de tráfico *broadcast* y tráfico de acumulación. Esta red no se describe en esta memoria.

En el presente trabajo se mejora la red de interconexión. Por ello, en las siguientes secciones se describe con mayor detalle la infraestructura de red de PEAK.

3.2 La Interfaz de Red

Esta unidad, también conocida como *NI* tiene como objetivo enviar y recibir los mensajes dentro del *tile* procedentes tanto de memoria principal, como de las caches (L1D, L2) y del *core* para enviarlas por las redes de PEAK (VN0, VN1, VN2, NIDs). Esta unidad debe también transferir mensajes entre elementos locales al *tile*. La Figura 3.4 nos muestra los módulos internos que contiene. En concreto, el módulo implementa cuatro tipos de unidades:

- Módulos *tonet*. Estos módulos están conectados a los diferentes elementos que pueden generar mensajes (cualquier tipo de mensaje) con destino a elementos locales al *tile* o externos. Así, se dispone de 7 elementos *tonet* dando soporte al controlador de memoria (MC), a la cache de instrucciones del *core* (L1I), a la cache de datos del *core* (L1D), al banco de cache de la L2 (L2), a la interfaz de acceso al TILEREG por parte del *core* (CORE), y al TILE-

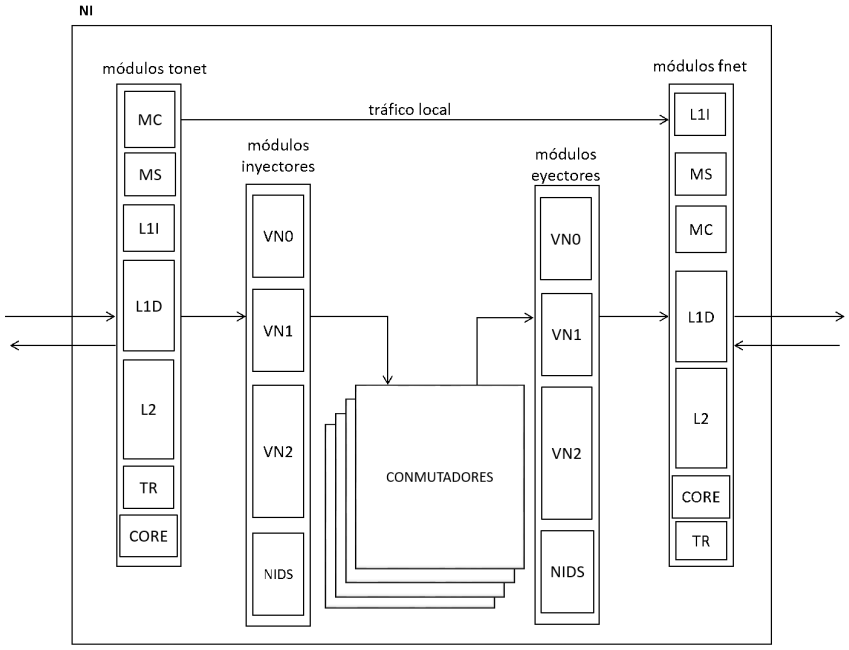

Figura 3.4: Estructura interna de la interfaz de red

REG (TR). Cabe destacar el módulo MS el cual implementa un generador de tráfico sintético que se utilizará posteriormente. Por tanto, este módulo también incorpora un módulo *tonet*.

- Módulos *fnet*. Estos módulos implementan la recepción de mensajes a cada uno de los posibles elementos receptores dentro del *tile*. En este caso también se definen siete módulos *fnet* para dar soporte al controlador de memoria (MC), la cache de instrucciones del *core* (L1I), la cache de datos del *core* (L1D), el banco de cache de la L2 (L2), la interfaz de acceso al TILEREG por parte del *core* (CORE), el TILEREG (TR) y el generador de tráfico sintético (MS).

- Módulos *inject*. Estos módulos inyectan mensajes a la red, por tanto, gestionan el tráfico no local del *tile* y están conectados directamente a los conmutadores del *tile*. En concreto, se implementa un módulo *inject* por cada red en PEAK: VN0, VN1, VN2, NIDs.

- Módulos *eject*. Estos módulos reciben mensajes externos al *tile*, y por tanto se conectan a los conmutadores del *tile*. De la misma forma que con los módulos *inject*, también se implementa un módulo *eject* por cada red en PEAK: VN0, VN1, VN2, NIDs.

Los módulos anteriores, 22 en total, se interconectan entre si con patrones diferentes tanto en conexionado como en amplitud de buses. En concreto, las conexiones entre módulos se refleja en la Tabla 3.1. Como puede verse en la tabla, no todas las conexiones entre módulos son posibles. Esto depende de la arquitectura

de PEAK y cómo gestiona las diferentes redes físicas. Así, por ejemplo, podemos ver como los módulos *tonet* de las unidades asociadas a L1I, TR, CORE, se conectan exclusivamente con la red VN2 y no con las redes VN1 ni VN0. De la misma forma las amplitudes de buses son muy distintas, estando en función del tamaño de datos a transmitir. Nótese que los bloques de cache en PEAK son de 64 bytes (512 bits) y que las direcciones son de 32 bits. También, el tamaño de *flit* en PEAK se define de 64 bits, pudiendo utilizar tamaños menores de *phit*.

fuente \ destino	MC_fnet	MS_fnet	L1I_fnet	L2_fnet	TR_fnet	CORE_fnet	L1D_fnet	VN0_inject	VN1_inject	VN2_inject
MC_tonet			x							x
MS_tonet		x						x		
L1I_tonet	x									x
L2_tonet	x						x	x	x	x
TR_tonet						x				x
L1D_tonet				x				x	x	
CORE_tonet					x					x
VN0_eject		x		x			x			
VN1_eject				x			x			
VN2_eject	x		x	x	x	x				

Tabla 3.1: Interconexiones entre módulos dentro de la interfaz de red

En este trabajo la interfaz de red (NI) se modificará sustancialmente, simplificando en gran medida los módulos de inyección (*inject*).

3.3 Red de Interconexión en PEAK

Los mensajes en PEAK se clasifican en tres tipos.

- *Mensaje corto de control*. Los mensajes cortos de control tienen un tamaño de un solo *flit* (el tamaño de *flit* en PEAK es de 64 bits). Estos mensajes llevan información de control entre los módulos TILEREG y el *core*.

- *Mensaje corto de datos*. Los mensajes cortos de datos son también de un solo *flit* y llevan la información de peticiones y respuestas entre los demás módulos de PEAK, principalmente entre las cache L1Ds, L2s y el controlador de memoria. Estos mensajes de datos suelen llevar la dirección de memoria que se quiere gestionar así como la petición a realizar (lectura, escritura, invalidación, ...).

- *Mensaje largo de datos*. Los mensajes largos de datos son mensajes que llevan 8 *flits* adicionales de datos, que coincide con el tamaño de bloque del sistema de caches de PEAK. Por tanto, estos mensajes son de 9 *flits* en total.

PEAK implementa conmutación *wormhole*, por tanto, la unidad básica de control de flujo es el *flit*. Esto quiere decir que pueden existir bloqueos temporales de mensajes largos que ocupan colas en conmutadores adyacentes. Ahora bien, el uso del algoritmo de encaminamiento en PEAK garantiza ausencia de bloqueo (*deadlock*).

En la Figura 3.5 se ilustra uno de los puertos del conmutador básico en PEAK. En concreto la lógica asociada al puerto de entrada del norte y el puerto de salida del norte.

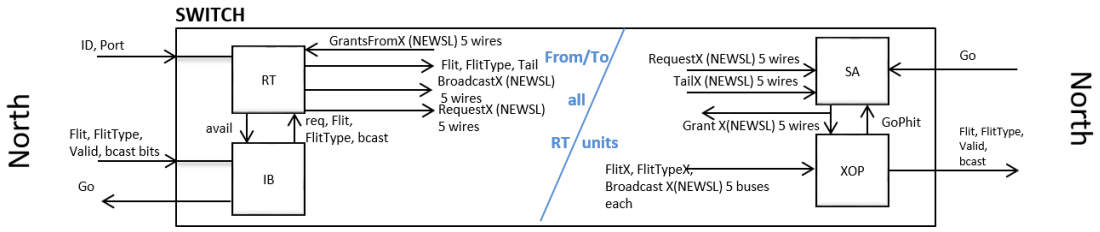

Figura 3.5: Estructura interna del conmutador representada en el puerto norte

Como podemos ver en la figura, el conmutador posee cuatro módulos diferenciados, dos asociados a cada puerto de entrada y dos asociados a cada puerto de salida. Cada puerto de entrada y salida implementa los mismos módulos.

El primer módulo es la cola de entrada (IB; *input buffer*). Este módulo se encarga de almacenar los *flits* que se reciben por el puerto de entrada y realizar la petición de encaminamiento al siguiente módulo. Además, este módulo implementa el control de flujo Stop & Go, generando una señal *go* hacia el puerto de salida asociado al puerto de entrada. El módulo IB implementa en PEAK solamente una única cola, por consiguiente no soporta canales virtuales ni ofrece ninguna solución de calidad de servicio.

El siguiente módulo en el conmutador (en el puerto de entrada) es el módulo de encaminamiento (RT; *routing*). Este módulo se encarga de calcular el puerto de salida del mensaje entrante. Para ello, se utiliza el algoritmo de encaminamiento XY, codificado con el mecanismo LBDR [13] implementado en dicho módulo. A su vez, el módulo RT almacena un *flit*, el cual será enviado al puerto de salida una vez se tenga acceso a él. Por tanto, todos los mensajes que entran por un puerto de entrada atraviesan los módulos IB y RT.

En el lado de salida del conmutador tenemos dos módulos: SA y XOP. El módulo SA (*switch allocator*) implementa el arbitraje en el acceso al puerto de salida. Cada puerto de entrada tiene una línea de petición (*request*) al módulo SA. El módulo implementa un arbitraje *round-robin* entre todas las peticiones de entrada, y

concede solamente un acceso (*grant*) a un puerto de entrada. Este arbitraje se realiza a nivel de mensaje. Solamente los *flits* cabecera de mensaje pueden generar una petición al árbitro. El recurso (puerto de salida) queda asignado a un puerto de entrada para todo el mensaje. Además, el árbitro tiene en cuenta el control de flujo del puerto asociado. Si la señal *go* que recibe no está activada, entonces el árbitro no concede ninguna petición.

El módulo XOP implementa un multiplexor con cola para seleccionar el *flit* de salida por el puerto. Este multiplexor se configura por el módulo SA asociado al puerto de salida. El módulo XOP recibe el *flit* de cada puerto de entrada.

Como se ha descrito anteriormente, PEAK implementa cuatro redes paralelas. Las dos primeras redes de interconexión en PEAK son idénticas, en el sentido que utilizan los mismos recursos y estrategia en sus componentes. La tercera red es ligeramente distinta ya que codifica el algoritmo de encaminamiento XY sin utilizar la estrategia LBDR.

Como podemos deducir de la descripción anterior, la red actual de PEAK (nos referimos principalmente a las redes VN0 y VN1) no soportan ningún mecanismo que garantice calidad de servicio (QoS). Podemos decir que la red actual en PEAK ofrece un servicio *best-effort* a las aplicaciones que en ella se estén ejecutando. El objetivo del presente trabajo reside en modificar la red de PEAK (principalmente VN0 y VN1) añadiendo un mecanismo que ofrezca calidad de servicio al sistema. En el siguiente capítulo describimos las modificaciones realizadas para obtener tal fin.

CAPÍTULO 4

Arquitectura PEAK con Soporte a Calidad de Servicio

En este capítulo se muestran las modificaciones realizadas en la arquitectura para proveerla de calidad de servicio por medio del uso de redes virtuales (VN) y canales virtuales (VC). Se describe con detalle cada uno de los componentes que se ha diseñado, modificado e implementado, así como la forma en que se encuentran interconectados entre sí para obtener el funcionamiento esperado.

4.1 Estrategia de Calidad de Servicio para PEAK

Como se ha comentado en el capítulo 3, nos encontramos con una arquitectura formada por varios *tiles* interconectados por medio de diferentes redes de conmutadores. Para dar soporte a calidad de servicio en la red de interconexión de datos se ha rediseñado la red de interconexión en PEAK para proveer una única red que soporte tanto la definición de redes virtuales como de canales virtuales. Las redes virtuales se encuentran aisladas de forma lógica y por lo tanto ningún mensaje de una red virtual puede terminar en otra, de este modo podemos garantizar el perfecto aislamiento de la información entre redes. Por otro lado los canales virtuales nos permiten tener una mayor utilización de los puertos del conmutador puesto que si un mensaje que viene por un puerto no puede avanzar, éste permanece en su canal y otro lo puede adelantar a lo largo de todo el *pipeline* del conmutador utilizando otro canal virtual diferente.

El objetivo es tener un sistema altamente configurable, siguiendo la estrategia de PEAK, donde en tiempo de síntesis e implementación el usuario pueda definir cuantas redes virtuales y cuantos canales virtuales va a necesitar, así como el tipo de gestión que se realice de dichos recursos. Para ello, se han definido nuevos parámetros globales en el fichero de configuración de PEAK que permiten definir el número de instancias de cada componente así como las longitudes va-

riables de los vectores según el número de canales virtuales y redes virtuales que se implementan en el conmutador. Todos los parámetros que se han añadido se encuentran enumerados en la Tabla 4.1.

Parámetro	Definición
VC_SUPPORT	Habilita el código Verilog que define el soporte para canales virtuales y redes virtuales. Si permanece comentado, el conmutador no implementará canales virtuales ni redes virtuales.
NUM_VN	Número de redes virtuales.
NUM_VC	Número de canales virtuales.
VC_DYNAMIC_WAY	Determina de que forma se asignarán los VC que queden por asignar a los mensajes nuevos. Este modo asigna equitativamente con un árbitro *round-robin* de prioridades fijas los VCs libres.
VC_STATIC_WAY	Determina de que forma se asignarán los VC que queden por asignar a los mensajes nuevos. Este modo asigna estáticamente los VC por lo que los mensajes entran y salen por el mismo VC.
VN_WITH_PRIORITIES	Activa el modo de prioridades por redes virtuales. El vector de prioridades es el que las definirá en términos de anchos de banda.
VN_WEIGTH_PRIORITIES_2	Vector de pesos para dos VN. De esta forma podemos asignar distintos pesos a las VN.

Tabla 4.1: Parámetros globales adicionales para soporte a redes y canales virtuales en PEAK.

Para ejercer la función de reserva de anchos de banda para diferentes redes virtuales se utiliza el vector de pesos en los árbitros de puerto. Este vector, consta de diez partes en las cuales se reparte el 100 % del ancho de banda. En cada una de estas partes se le asigna la máxima prioridad de salida a una de las redes virtuales con el fin de garantizar un ancho de banda en función de los pesos asignados en el vector. A continuación se muestra un ejemplo del vector de pesos en el que se definen prioridades de un 20 % para la VN0 y de un 80 % para la VN1.

```
‘define VN_WEIGTH_PRIORITIES_2 1’d1,1’d1,1’d1,1’d0,1’d1,1’d1,1’d1,1’d0,1’d1,1’d1
```

En la Figura 4.1 podemos observar los cambios a nivel de *tile* realizados. Al introducir los cambios en PEAK se percibe una reducción del número de conmutadores. En concreto, las dos redes de datos de PEAK (VN0 y VN1) se han unificado en una única red de datos con soporte para VNs y VCs. Hay que tener en cuenta que toda la información que pasaba antes por dos redes físicas distintas, ahora pasa a través de una sola red con el consiguiente incremento en la

utilización de los puertos físicos de cada conmutador. Las dos redes restantes, VN2 y NIDs, no se han modificado ni alterado en los nuevos diseños. En un trabajo futuro se integrarán estas redes también en la nueva red con soporte para VNs/VCs.

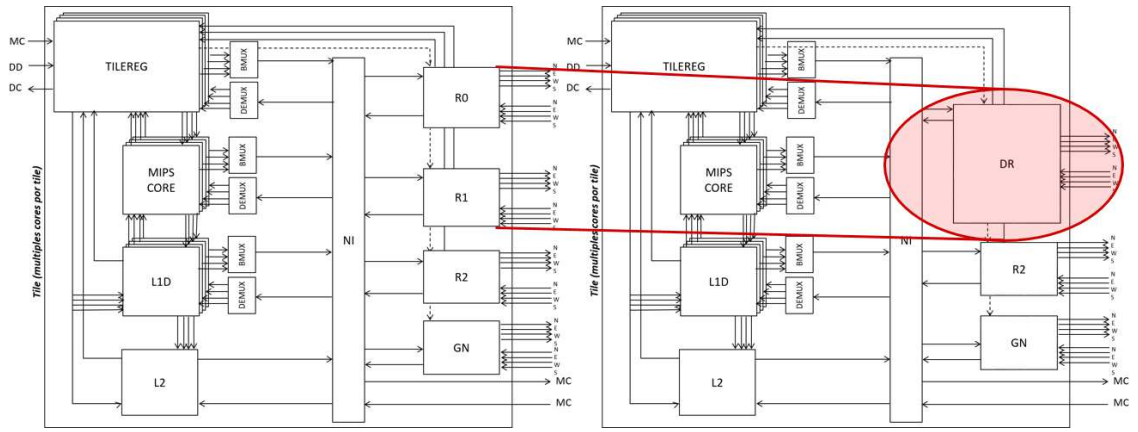

Figura 4.1: Cambios realizados en el *tile*

Por otro lado, a causa de los cambios en las redes de datos, se han modificado y añadido módulos en el interfaz de red (NI). La Figura 4.2 representa las modificaciones que se han llevado a cabo. Puesto que las redes de datos se han unificado en una misma red física, la lógica de los inyectores anteriores se ha unido igualmente formando una nueva unidad denominada INJECT. Si observamos la figura, además de este componente aparecen otros nuevos componentes que son unos serializadores. Estos componentes aparecen para simplificar la lógica del inyector. El objetivo principal de estas unidades es serializar los mensajes que provienen de los módulos *tonet* en partes de 64 bits (tamaño del *flit* actual) al módulo INJECT.

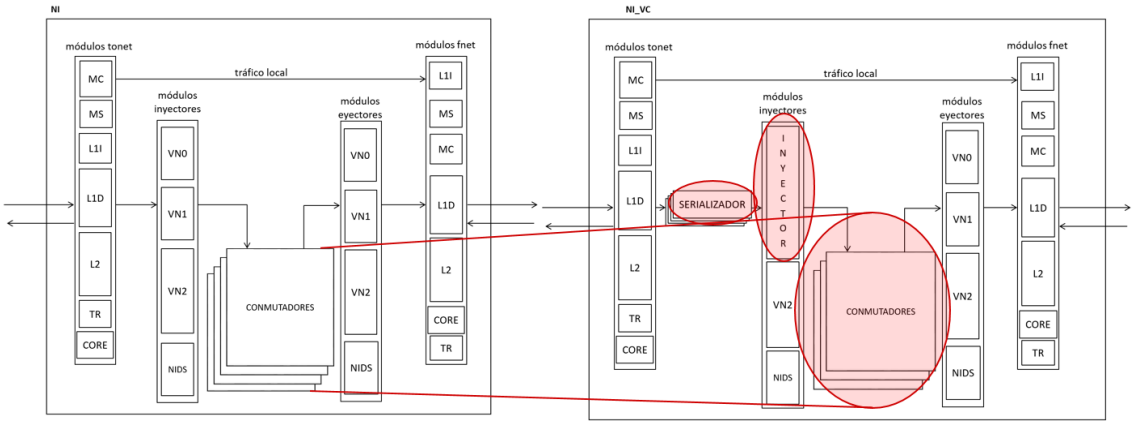

Figura 4.2: Cambios realizados en la interfaz de red

Nótese que con el nuevo diseño la red tiene soporte para redes virtuales y canales virtuales, lo cual nos llevará a una mayor utilización y gestión de los enlaces físicos. Pero ahora bien, dado que la red de datos inyecta y extrae los datos

a través de los puertos locales de los conmutadores, si estos puertos locales no los proveemos de la misma estrategia, la productividad y efectividad en QoS de la red no se incrementará ya que los inyectores y eyectores se convertirán en el cuello de botella del sistema de red. Para que esto no ocurra, se ha provisto al módulo inyector de una lógica muy parecida a la que contienen los conmutadores.

El conmutador es uno de los componentes principales de la red de interconexión, puesto que es el que nos permite el paso de la información entre *tiles*. Para representar los cambios realizados nos centraremos únicamente a nivel de un puerto. En la Figura 4.3 se encuentran representados los cambios de señales y módulos a lo largo de todo el *pipeline* del conmutador, para la lógica asociada al puerto de entrada del norte y el puerto de salida del norte. La estructura interna del conmutador ha variado notablemente puesto que este elemento es el principal para implementar el soporte de canales virtuales y redes virtuales. Con la nueva implementación, el conmutador está formado por cinco módulos, algunos de ellos asignados al puerto de entrada y otros asignados al puerto de salida. De hecho, todos los módulos están instanciados para cada puerto (de entrada o de salida) del conmutador. Los módulos en cuestión son: IB (*Input Buffer*), RT (*Routing*), VA (*Virtual Channel Allocator*), SA (*Switch Allocator*) y XOP (*Crossbar Output*). En cada puerto de entrada del conmutador se soportan varios canales virtuales. Actualmente se soportarán dos redes virtuales por lo que los canales se repartirán a partes iguales entre las redes virtuales, la mitad inferior pertenecerá a la red virtual 0 (VN0) y la mitad superior a la red virtual 1 (VN1).

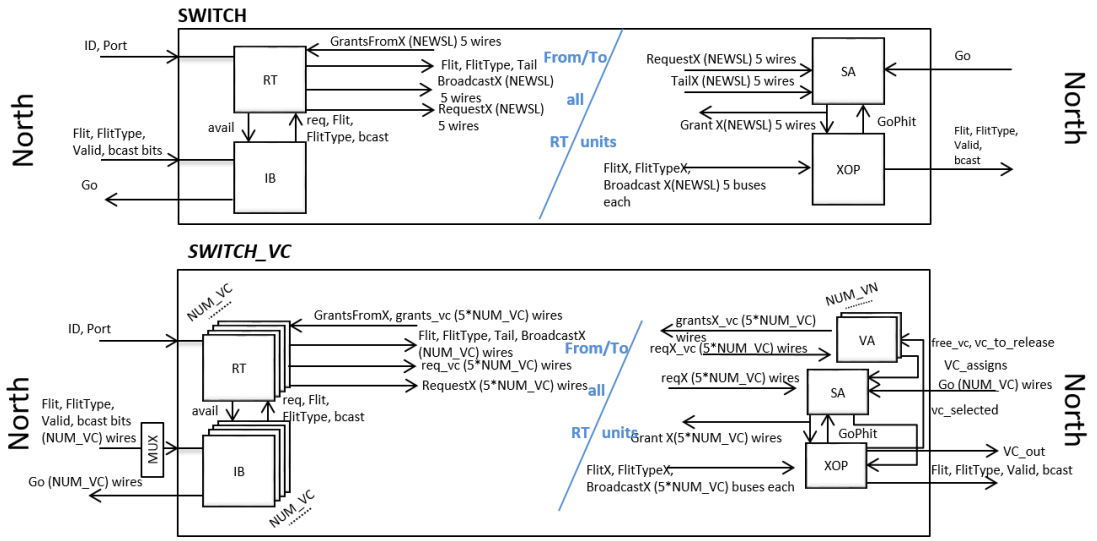

Figura 4.3: Cambios realizados en el conmutador representados en el puerto norte

El módulo IB (*Input Buffer*) es una cola situada en la primera etapa del *pipeline* del conmutador y se utiliza para almacenar los *flits* que se reciben, mientras la unidad RT (*Routing*) encamina la cabecera de cada mensaje y pide las peticiones de salida a los bloques VA y SA. El módulo VA (*Virtual Channel Allocator*) es el encargado de recibir peticiones de la unidad RT y asignarles un canal virtual que

se encuentre libre. El SA (*Switch Allocator*) es el encargado de conceder o no los permisos de salida a los *flits* de los mensajes por el puerto de salida asociado. En la última etapa del *pipeline* se encuentra la unidad XOP (*Crossbar Output*).

El proceso que siguen los paquetes a lo largo del *pipeline* es el siguiente. Inicialmente los paquetes llegarán a las colas y se almacenarán en ellas. Mientras tanto, la cabecera de los mensajes pasará a la unidad de encaminamiento para calcular el puerto de salida y el canal virtual a utilizar en el siguiente salto. De hecho, la unidad de RT realizará la petición de obtención de canal virtual a la unidad VA. Una vez obtenido, la unidad RT realiza una petición a la unidad SA por cada *flit* del mensaje. La unidad SA, al aceptar una petición, notifica a la unidad RT correspondiente y configura la unidad XOP asociada con el puerto de salida.

Un primer cambio significativo es la instanciación de múltiples módulos IB y RT en cada puerto de entrada, como consecuencia del soporte a múltiples VNs/VCs. En concreto, instanciamos tantos módulos como canales virtuales totales dispongamos. Otro cambio significativo es la presencia de un nuevo módulo, el árbitro de canales virtuales (VA) que se instancia, en cada puerto de salida, tantas veces como redes virtuales (VNs) se estén implementando. Nótese que los módulos SA y XOP se instancia solo una vez en cada puerto de salida. Ahora bien, todos estos módulos han sufrido grandes cambios tanto internos como de conexionado entre ellos.

Para obtener las máximas presentaciones se ha decidido instanciar un módulo VA por cada red virtual. Los mensajes de una red virtual solo pueden solicitar recursos de esa misma red virtual y este proceso se puede realizar en paralelo entre diferentes redes virtuales. Nótese, sin embargo, que el interconexionado entre módulos aumentará de forma considerable y por lo tanto la lógica de los componentes que ya se encontraban en la arquitectura PEAK también aumentará de forma notable así como su complejidad en el diseño. Por ejemplo, la señal *VC_out* de la figura indicará en cada puerto de salida que canal virtual se está utilizando. Por otra parte, en la versión anterior la señal *go* de cada puerto de salida era un cable y en la actual será un bus. Esto es debido a que sólo se gestionaba un IB por puerto de salida y ahora como hemos visto habrá tantos IB en la entrada de cada puerto como canales virtuales se deseen soportar. En la siguiente sección se entra en mas detalle sobre cada una de las unidades que componen el nuevo conmutador con soporte a QoS.

4.2 Componentes Internos del Conmutador

Seguidamente se detalla el diseño interno de los módulos que hacen posible el mecanismo de los canales y redes virtuales, y por consiguiente el soporte inicial de QoS.

4.2.1. La Cola de Entrada (IB)

Este componente se encuentra situado en los puertos de entrada del conmutador. Se instancia uno por cada canal virtual que se soporte en cada puerto. Su función principal es encolar los *flits* de los mensajes entrantes. Para ello, contiene tres colas circulares para almacenar los *flits* entrantes, el tipo de *flit* (cabecera, cuerpo y cola) y si son *flits* de *broadcast* o no. Tal y como se muestra en la Figura 4.4, cada cola tiene sus punteros de lectura y escritura correspondientes. La señal de control de flujo STOP & GO depende de los marcadores superiores e inferiores de las colas que determinan cuando se detiene o se reanuda el flujo de *flits* entrante. Para llevar la cuenta de los *flits* almacenados se utiliza un registro que contabiliza los *flits* encolados en cada momento.

El módulo IB tiene varios puertos de entrada y de salida definidos. La señal *Valid* cual indica si la información de entrada es válida o no. El puerto de entrada *Flit* contiene el *flit* que se acaba de recibir y la señal *FlitType* codifica el tipo de *flit* entrante que pueden ser del tipo: Cabecera, cuerpo, cola o cabecera y cola (mensajes de tamaño de un solo *flit*). El puerto *BroadcastFlit* indica si el *flit* que se recibe es un *flit* de tipo *broadcast* o no. La señal *Req_RT* se activa cuando se envía un *flit* a la unidad RT para notificar al módulo de encaminamiento. El puerto *Avail* indica que la unidad RT está disponible para recibir un *flit*. Los puertos de salida (*FlitOut, FlitTypeOut y BroadcastFlitOut*) se utilizan para enviar el *flit* a las unidades de encaminamiento RT. Por último, la señal (*Go*) activa o desactiva el flujo de *flits* destinados a esta unidad.

El módulo IB no se ha modificado para la nueva arquitectura. Ahora bien, para una correcta gestión de los canales virtuales, se ha añadido un demultiplexor en el puerto de entrada, el cual selecciona el módulo IB que recibirá el *flit*. También se han agrupado las señales *go* que genera cada módulo IB, para ser enviadas al puerto de salida asociado en el conmutador anterior. Nótese que cada módulo IB tendrá asociado un módulo RT distinto, por lo que el interfaz entre ambos módulos no se ha modificado.

4.2.2. La Unidad de Encaminamiento (RT)

La unidad RT se instancia por cada canal virtual que se soporte en cada puerto y tiene la función principal de calcular el puerto de salida por el que se debe enviar el mensaje actual. Para ello hace uso del algoritmo de encaminamiento XY. Ahora bien, el módulo RT recibe el *flit* desde el módulo IB asociado y es el encargado de solicitar los recursos adecuados y de transmitir el *flit* hacia el puerto seleccionado. Por tanto, las funciones del módulo RT son:

- Realizar el encaminamiento de los *flits* de cabecera.
- Solicitar un canal virtual (al módulo VA) para un *flit* de cabecera.

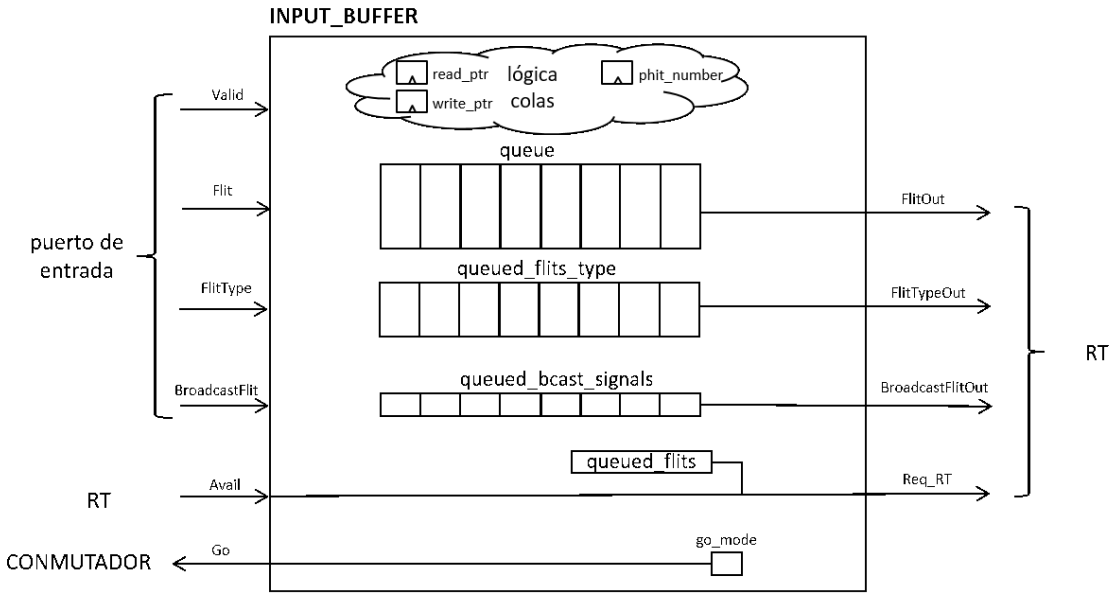

Figura 4.4: Estructura de la cola de entrada

- Solicitar el puerto de salida (al módulo SA) para cada *flit* del mensaje que se reciba.

Según se representa en la Figura 4.5, al módulo RT recibe puertos de tres tipos provenientes de la unidad IB. El bus *Flit* contiene los datos del *flit*, *FlitType* el tipo de *flit* y *BroadcastFlit* indica cuando el *flit* es del tipo *broadcast*. La señal *Req* se activa cuando llega información del IB. El conmutador informa a este módulo del puerto *Port* y *tile Cur* en el que se encuentra ubicado. Los árbitros de canales virtuales y los de puertos de salida de cada puerto notifican mediante las señales *Grant_x* cuando ha sido concedida alguna de las peticiones provenientes de este puerto y este canal virtual. Las peticiones a los árbitros se realizan mediante las señales de salida *Request_x*. Cuando alguna petición esta lista para abandonar el puerto se envía su información a la unidad XOP mediante los buses referentes al *flit*, el tipo de *flit* y si es un *flit* de *broadcast*. El módulo contiene registros que almacenan la información del *flit* que actualmente se está gestionando, su tipo y si es del tipo *broadcast*. En esta unidad también se implementan registros para almacenar si se encuentra algún canal asignado (*VC_assigned_X*) o si se le ha concedido alguna petición por un puerto de salida anteriormente (*Xant*).

La unidad RT funciona de la siguiente forma. Inicialmente almacena el *flit* de entrada en una cola, así como su tipo y si es de tipo *broadcast*. Este almacenamiento es necesario puesto que toda cabecera de un mensaje debe realizar primero una petición de canal virtual y una vez se le ha concedido debe realizar peticiones de salida por el canal físico. Las peticiones de canal virtual solo se realizan una vez (las cabeceras) pero las peticiones del puerto de salida se realizan por cada *flit*. Esta es una gran diferencia con la arquitectura PEAK anterior, ya que al no disponer de canales virtuales dos mensajes no podían ser transmitidos de for-

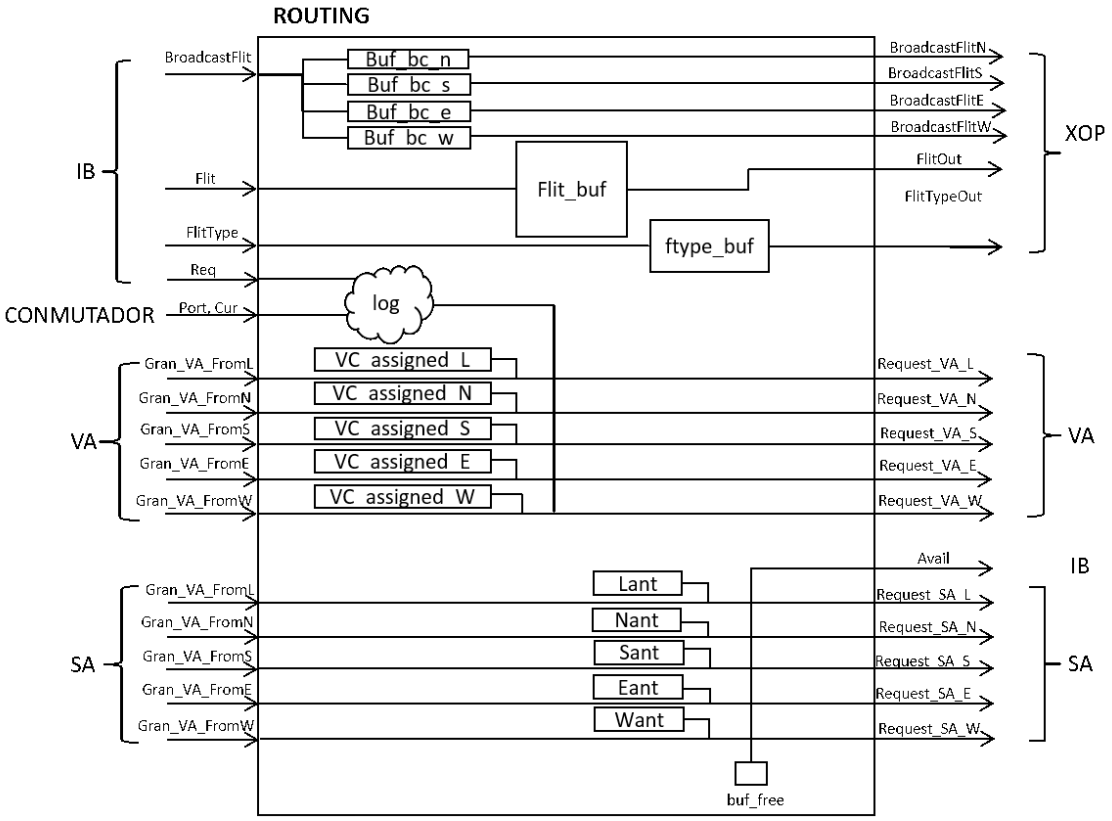

Figura 4.5: Estructura de la unidad de encaminamiento

ma intercalada. Por tanto, en la versión PEAK original tenemos multiplexación a nivel de mensaje y en la nueva versión tendremos multiplexación a nivel de *flit*.

Si nos fijamos en la figura, percibimos unos registros denominados *VC_assigned_x* que notifican cuando el *flit* de cabecera de un mensaje tiene un canal virtual asignado. En caso de estar asignado, el módulo RT procede a la petición del canal físico. Existen otros registros de estado *Xant* que notifican cuando el puerto de salida ha concedido un permiso de salida al menos a la cabecera del mensaje y en este caso los siguientes *flits* seguirán realizando peticiones al mismo puerto de salida.

El principal cambio realizado en el módulo RT versa sobre la conexión con el nuevo módulo VA y su gestión interna. También es nuevo el automata que discrimina entre el acceso al módulo VA o al módulo SA en función del tipo de *flit* y del estado interno del módulo.

4.2.3. El Árbitro de Canales Virtuales (VA)

El módulo VA tiene la función de asignar un canal virtual a cada una de las peticiones que le lleguen. Las peticiones de las señales de entrada provienen de los módulos de encaminamiento correspondientes a cada puerto de entrada y

por tanto, pueden llegar varias peticiones de puertos de entrada diferentes y de varios de sus canales virtuales al mismo tiempo (en el mismo ciclo). Cada red virtual VN dispone de un módulo VA independiente.

Este componente se ha implementado de dos formas diferentes. En una primera versión, los canales virtuales disponibles para una red virtual pueden ser asignados de forma dinámica entre todas las peticiones en la entrada. La asignación se realiza de forma equitativa por medio de un árbitro *round-robin*. En una segunda versión, los canales virtuales asociados a una red virtual son asignados de forma estática, de forma que una petición procedente de un canal virtual x en la entrada únicamente podrá optar por el canal virtual x en el siguiente conmutador. De esta forma se permiten más asignaciones por unidad de tiempo, incluso en la misma red virtual.

4.2.3.1. El VA con Asignación Dinámica

A continuación se describe la implementación del árbitro de canales virtuales con asignación dinámica. Si prestamos atención a la Figura 4.6, el tamaño de los puertos de entrada y salida, así como los recursos internos están en función del número de canales virtuales que se soportan por red virtual.

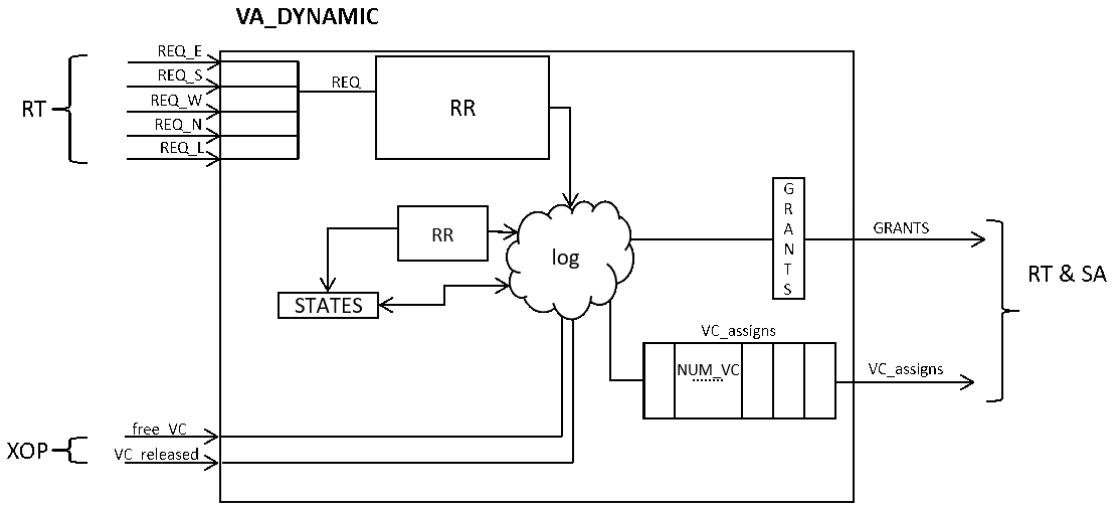

Figura 4.6: Estructura del árbitro de canales virtuales con asignación dinámica

Como ejemplo, supongamos que el conmutador tiene 5 puertos de entrada y soporta 4 canales virtuales por red virtual. Con esta configuración, cada puerto de entrada tendrá un bus de cuatro bits con las peticiones del puerto de entrada *REQ_X*. En total tendremos 20 lineas de petición. El módulo generará a su vez 20 líneas de acceso *GRANTS*, repartidas en cuatro líneas por puerto de entrada. El módulo también incluye un vector de asignaciones de canales virtuales *VC_assigns* que identifica cada canal virtual gestionado con la entrada que lo tiene asignada. Por tanto, este recurso tiene un tamaño de $4 \times x$, donde x es el número

de bits necesarios para representar 20 valores (4×5). Por último, el módulo posee dos entradas para notificación de la liberación de un canal virtual. En concreto, una señal que comunica si un canal se libera *free_VC*, y un bus que indica el canal virtual liberado *VC_released*. Finalmente, el módulo incluye un vector de estados *STATES* que indica cuando un canal virtual está ocupado o se encuentra libre.

El módulo opera de la siguiente forma. En cada ciclo las peticiones de entrada al módulo se filtran a través de un árbitro *round-robin*, el cual elige una única petición. Una vez se conoce la petición elegida, se busca en el vector de asignaciones un canal virtual que esté disponible. Si es el caso, se asigna el canal virtual a la petición y se envía el grant al puerto de entrada. En el instante que un mensaje sale por completo por el módulo XOP, éste le comunica al VA el canal virtual para que libere el estado del canal asignado al mensaje.

La liberación y reasignación de canales virtuales se ha optimizado en el diseño, de tal forma que en el mismo ciclo puede ocurrir que un canal virtual se libere y se asigne de nuevo a otra petición de entrada. Esta optimización es muy importante cuando el número de canales virtuales por red virtual es bajo. Por ejemplo, con un solo canal virtual por red virtual, asignaciones del mismo canal virtual a diferentes mensajes conllevaría, sin la optimización, a ciclos de pérdida donde el canal virtual no fuera asignado. Con la optimización, incluso con un solo canal virtual por red virtual, el canal virtual se asigna a diferentes mensajes sin ciclos de pérdida, obteniendo un 100 % del rendimiento. Para obtener tal comportamiento, el vector de estados de los canales virtuales se ha combinado con las señales de liberación de canal virtual en la lógica que implementa la asignación de canales virtuales a peticiones de entrada.

4.2.3.2. El VA con Asignación Estática

En el segundo tipo de módulo VA, las asignaciones se realizan de forma estática (un mensaje siempre utiliza el mismo identificador de canal virtual en toda su ruta). El árbitro se muestra en la Figura 4.7. Tal y como se ha mencionado anteriormente, esta implementación es capaz de realizar varias asignaciones a diferentes canales virtuales en el mismo ciclo de reloj. De hecho, todas las asignaciones de canales virtuales son concurrentes.

En la implementación anterior con asignación dinámica se implementa un árbitro *round-robin* al cual le llegan todas las peticiones de entrada. Sin embargo, en el árbitro con asignación estática, las peticiones de entrada se reagrupan por canales virtuales y las que provienen del mismo canal compiten entre ellas por el mismo recurso en un árbitro *round-robin*. Por tanto, se implementan tantos árbitros RR como canales virtuales se soporten y cada canal tiene un registro que contiene el estado en el que se encuentra, ocupado o libre.

El vector de asignaciones de la implementación anterior también se utiliza en el árbitro con asignación estática. Cada canal virtual debe identificar qué entrada lo tiene asignado. Ahora bien, la lógica de asignación y desasignación está des-

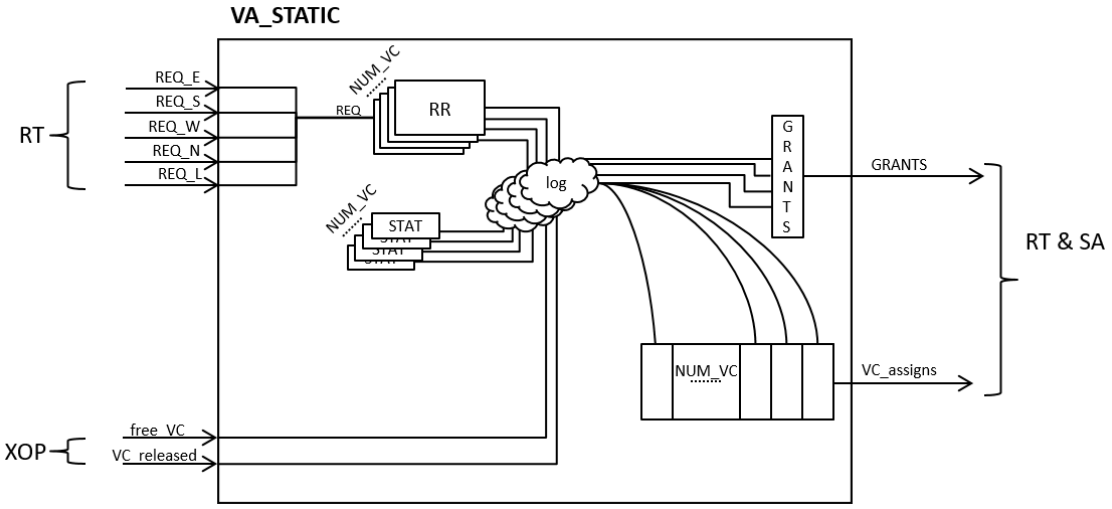

Figura 4.7: Estructura del árbitro de canales virtuales con asignación estática

acoplada para cada canal virtual ya que no todas las peticiones pueden competir por todos los canales virtuales.

4.2.4. El Árbitro de Puerto (SA)

El módulo SA tiene la misión de conceder el acceso de salida por el puerto asociado de entre todas las peticiones de entrada. Las peticiones, al igual que en el módulo anterior, provienen de los módulos de encaminamiento RT y por lo tanto podrán llegar en el mismo ciclo varias peticiones de cada puerto, una por cada canal virtual. Además, el módulo SA recibe peticiones provenientes de todas las redes virtuales. El módulo recibe como entrada el vector de asignaciones de los módulos VA (uno por cada VN), las señales de control de flujo provenientes de los módulos IB del puerto de entrada asociado al puerto de salida, así como la señal *GoPhit* que genera la unidad XOP. Este componente se ha implementado de dos formas diferentes. En una primera versión, las peticiones de entrada son seleccionadas equitativamente por medio de un árbitro *round-robin*. En una segunda versión, las peticiones entrantes son inicialmente separadas por VNs y después seleccionadas por prioridades.

4.2.4.1. El Árbitro de Puerto con Asignación Equitativa

A continuación se describe la implementación del árbitro de puerto con selección equitativa. Si prestamos atención a la Figura 4.8, el tamaño de los puertos de entrada y salida, así como los recursos internos están en función de los canales virtuales que se soportan.

El tamaño de los puertos de entrada y de salida está en función de los canales virtuales y redes virtuales que se soportan. Asumiendo un conmutador con 5

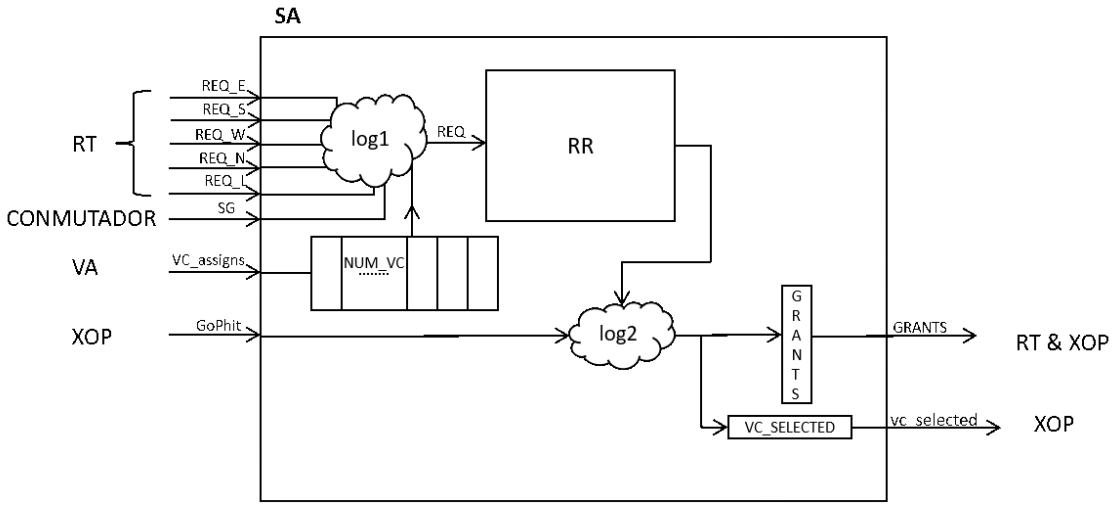

Figura 4.8: Contenido del componente switch allocator con asignación equitativa

puertos y con soporte para cuatro canales virtuales (dos canales virtuales y dos redes virtuales), cada puerto de entrada tendrá un bus de cuatro bits con las peticiones del puerto de entrada *REQ_X*. En total tendremos 20 lineas de petición. Al módulo también se le proporciona un vector de asignaciones proveniente del módulo VA. En este vector se encuentran identificados cada canal virtual gestionado con la entrada que lo tiene asignado. La señal *GoPhit* de entrada indica si el canal físico de salida se encuentra disponible. El módulo generará a su vez 20 lineas de acceso *GRANTS*, repartidas en cuatro lineas por puerto de entrada y otra señal que identifica el canal virtual correspondiente a la petición que se le ha concedido el acceso.

El módulo funciona de la siguiente forma. En cada ciclo de reloj las peticiones a la entrada son filtradas por una lógica inicial. Esta lógica solo permitirá proceder a aquellas peticiones si la señal de control de flujo del canal asignado está activada. Nótese que la señal *go* es la asociada con el canal virtual asignado a la petición. En el siguiente paso, un árbitro *round-robin* elige una única petición que se le asigna un *grant*. Adicionalmente se considera también la señal de *go* a nivel de phit (señal *GoPhit*).

Tal y como se observa en la figura, las señales de salida envían la petición que ha sido seleccionada satisfactoriamente a la unidad de encaminamiento y a la de salida en el crossbar (XOP).

4.2.4.2. El Árbitro de Puerto con Asignación por Pesos

En aras a mejorar esta versión del árbitro de puerto, se especifica su otra implementación con prioridades por pesos. En este módulo es donde se verá reflejada la función de reserva de anchos de banda para diferentes redes virtuales. Si observamos la Figura 4.9, los puertos de entrada y salida son totalmente idén-

ticos a la primera versión puesto que únicamente varía su estructura interna. El módulo contiene árbitros que gestionan las peticiones pertenecientes a cada red virtual por separado. Existe una lógica que ejerce la función de prioridades. Un vector de pesos (*WEIGHTS*) es el que designa los pesos a cada red virtual.

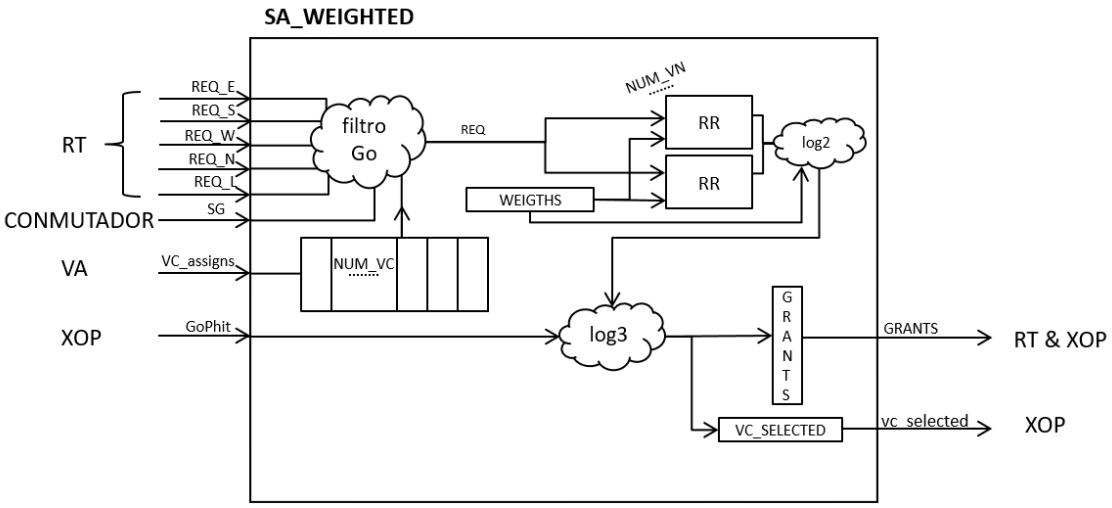

Figura 4.9: Contenido del componente switch allocator con prioridades

El módulo actuá de la siguiente forma. Igual que en la versión anterior las peticiones de entrada son filtradas inicialmente según las señales de control de flujo y los canales virtuales que se les han asignado en los árbitros de canales virtuales. En el siguiente paso se gestionan las peticiones de cada canal virtual por separado. La siguiente fase separa las peticiones por redes virtuales. A cada red se le asigna un árbitro *round-robin* diferente y se selecciona una petición por cada VN. Llegados a este punto, una lógica que tiene en cuenta el vector de prioridades elige una única petición que se le asigna un *grant*. Igual que en la anterior implementación se considera también la señal de *go* a nivel de phit (señal *GoPhit*).

El vector de pesos es un registro de 10 elementos. Cada elemento contiene un valor que identifica un VN. De esta forma, para el vector {0,0,0,1,0,0,0,1,0,0} se representa el VN0 ocho veces y 2 veces el VN1. Con este vector se conseguirá reservar un 80 % del ancho de banda del enlace al tráfico del VN0 y el restante al VN1. Cambiando el contenido del vector se obtiene una nueva distribución del ancho de banda del enlace. El vector de pesos se utiliza para discriminar que petición de VN0 o VN1 consigue finalmente el acceso al puerto. Para ello, un *token* cíclico recorre el vector de pesos. Este *token* avanza en cada ciclo que el árbitro concede un acceso. Si existe una petición en el VN indicado por el vector de pesos, entonces se selecciona. En caso contrario se concede la petición del otro VN, si existe.

Como se puede ver en la figura, se emiten señales notificando la petición que se le ha concedido el acceso a la unidad de encaminamiento y a la de salida en el crossbar (XOP).

4.2.5. El Crossbar de Salida (XOP)

El módulo XOP realiza la multiplexación de todos los *flits* dirigidos al puerto de salida asociado al módulo. A este módulo le pueden llegar *flits* de los diferentes puertos de entrada y desde cada puerto de entrada de cada uno de sus canales virtuales (incluso de diferentes redes virtuales). Para una correcta multiplexación, el módulo SA asociado al puerto facilita las señales de los GRANTS y el canal virtual seleccionado (*vc_selected*).

Si nos fijamos en la Figura 4.10 el tamaño de los puertos de entrada y de salida están en función de las redes virtuales y canales virtuales soportados. Si suponemos que el conmutador tiene 5 puertos de entrada y tiene soporte para 4 canales virtuales. Cada puerto de entrada tendrá un bus de cuatro *Flits* (uno por cada canal virtual) indicando el contenido de cada *flit*. Estas señales tendrán una longitud de 4× *Flit_size*, donde *Flit_size* es un parámetro global del sistema que indica el tamaño del *flit*. También existen buses indicando el tipo de *flit* y si el *flit* es de tipo *broadcast*. Estas señales tendrán una longitud de 4× *FlitType_size* y de 4 bits respectivamente. El módulo generará a su vez un bus por cada tipo referentes al *flit* de salida. También notifica la señal *Valid* que indica la validez de la información y el canal virtual asignado a ese *flit VC_out*. Los tres buses y la señal de *Valid* se envían por el puerto de salida del conmutador. Existen otras señales que se envían a los árbitros para gestionar el control a nivel de *phit*: *GoPhit* y *free_VC* para indicar cuando se libera un canal virtual y se le pueda asignar a otro *flit*.

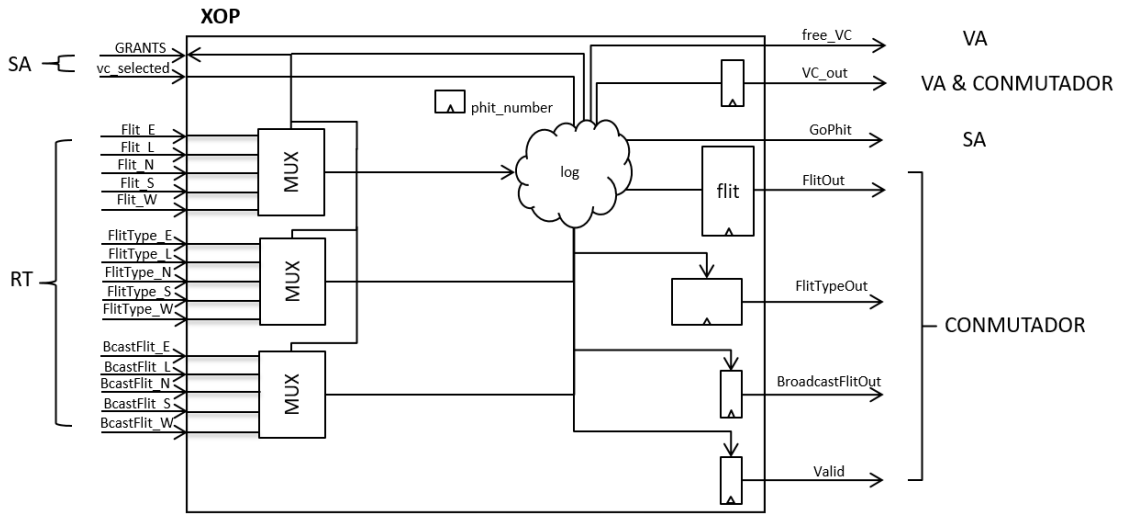

Figura 4.10: Contenido del componente XOP

Los buses de tres tipos que envía cada puerto se multiplexan según el bus *GRANTS* proveniente de la unidad SA. Una vez se han multiplexado las señales, estas salen por el puerto de salida correspondiente del conmutador indicando por medio de la señal *VC_out* el canal virtual que se le ha asignado.

En el caso que un mensaje termine de salir por completo a través de esta unidad, existen otras señales (*free_VC* y *VC_out*) de salida que indican a la unidad VA cuando un canal virtual queda libre.

4.2.6. El Árbitro *Round-Robin*

En los módulos anteriores de VA y SA se ha utilizado un árbitro *Round-Robin*. El árbitro ejerce la función de equidad en muchas partes del conmutador. Se le suministra un vector de peticiones (en cada ciclo de reloj) y el módulo es el encargado de seleccionar únicamente una de estas peticiones de forma rotatoria y equitativa.

Si nos fijamos en la Figura 4.11 el vector de peticiones se conoce como *REQ*. Como señales de salida este módulo devuelve el mismo vector que se le ha proporcionado pero con solo un bit activo *GRANTS*. Otra señal de salida *grants_id* indica la posición de la petición seleccionada en el vector.

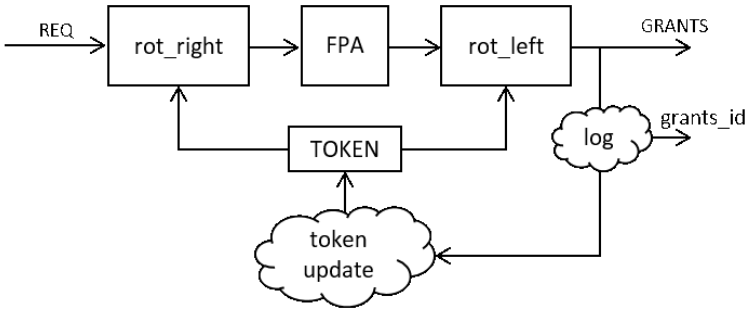

Figura 4.11: Contenido del árbitro *Round-Robin*

El funcionamiento del módulo lo realizan básicamente cuatro módulos que contiene. Inicialmente la unidad *token_update* indica a los otros módulos el *token* para este ciclo de reloj. Esta señal se calcula a partir de la última petición satisfactoria e indica cual es la petición con más prioridad. Seguidamente, la unidad *rot_right* realiza una rotación a derechas del vector de peticiones. La rotación será de tantas posiciones como el valor del *token* indique. Después de realizar la primera rotación, el componente (*FPU* o *Fixed-Priority-Arbiter*) que es un árbitro de prioridades fijas, selecciona una única petición con la posición más baja. Finalmente la unidad *rot_left* realiza una rotación con el mismo número de posiciones que la inicial, para que los bits del vector queden en las mismas posiciones. Nótese que al finalizar todas las fases el vector únicamente tiene activado un bit, que es el correspondiente a la petición seleccionada.

4.3 Componentes Adyacentes al Conmutador

El conmutador es el elemento central que implementa la provisión para calidad de servicio (QoS). Ahora bien, hay otros componentes, principalmente en el interfaz de red (NI) que deben ser modificados y adaptados a los nuevos requisitos de QoS. En esta sección ofrecemos una descripción de dichos componentes.

4.3.1. El Serializador de *Flits*

Con el objetivo de simplificar el diseño de los demás componentes, se ha creado un serializador de *flits*. Esta unidad divide un mensaje largo en varios *flits*. El módulo tiene un puerto de entrada por donde llega el mensaje completo junto con una señal que indica el número de *flits* que ocupa. A su vez, el módulo tiene un puerto de salida por donde emite de forma secuencial los *flits*.

Tanto en el puerto de entrada como en el puerto de salida se implementa el correspondiente control de flujo mediante las señales *avail* y *req*. La señal *avail* indica que la unidad dispone de espacio para almacenar un mensaje/*flit*. Con la señal *req* se notifica la recepción de que un nuevo mensaje/*flit*. La Figura 4.12 muestra el esquema del serializador.

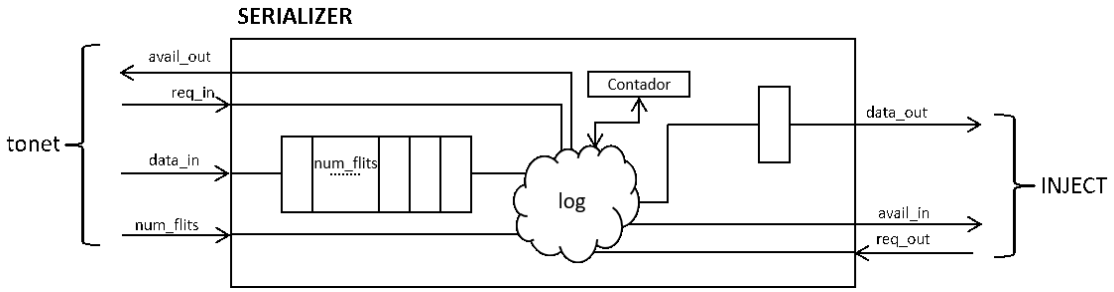

Figura 4.12: Estructura interna del serializador

4.3.2. El Inyector

El inyector es la unidad que inyecta los *flits* de los mensajes desde el *tile* al conmutador. Estos mensajes pueden provenir de diferentes fuentes (L1D, L2, ...). También puede darse el caso que los *flits* se inyecten por el generador de tráfico sintético que se ha implementado en cada *tile*. Cada una de estas fuentes puede inyectar por diferentes redes virtuales y, por lo tanto, los mensajes deben permanecer separados a lo largo de todo el trayecto por la red de interconexión hasta llegar a su nodo destino, incluido en el inyector.

En la Figura 4.13 se representa la estructura interna del inyector y como se puede observar se han implementado varias colas a la entrada. Estas colas nos

ayudan a una mejor gestión de las señales *avail* y *req*. Las colas deben tener una capacidad mínima de dos *flits* para obtener el máximo rendimiento en la inyección (*round-trip time*).

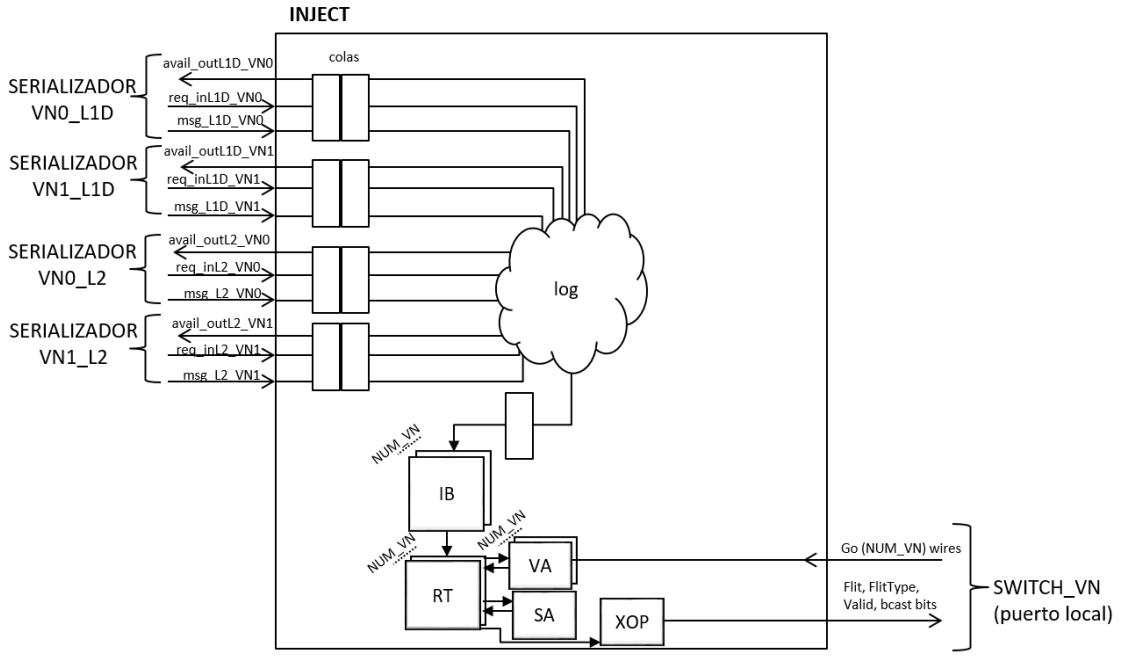

Figura 4.13: Lógica y estructura interna de la unidad de inyección

Existen diferentes cuestiones de diseño que deben ser tenidas en cuenta en la implementación del inyector. En primer lugar, debemos establecer un órden de prioridades entre las diferentes fuentes de tráfico. En concreto, las prioridades que hemos implementado son: primero los *flits* del generador de tráfico sintético (MS), seguidamente los que provienen de la L1D y finalmente los *flits* de la L2.

La asignación de fuentes de tráfico y redes virtuales debe realizarse de forma estática y atendiendo a las directrices de diseño de la arquitectura PEAK. En concreto, la Tabla 4.2 muestra la asignación de fuentes de inyección a redes virtuales definidas.

	VN0	VN1
MS	x	x
L1D_tonet0	x	
L1D_tonet1		x
L2_tonet0	x	
L2_tonet1		x

Tabla 4.2: Asignación de fuentes de inyección a redes virtuales definidas

Un *flit* al que se le permite acceder a su red virtual, pasa al módulo IB asociado a la red virtual. Este módulo implementa una cola circular que almacena

los *flits* que van a ser inyectados. Los *flits* almacenados en IB siguen su recorrido pasando por el módulo RT. Este módulo implementa la gestión con el árbitro VA y SA, de la misma forma que se realiza en los módulos RT del conmutador. De hecho, los módulos VA y SA son los mismos que se utilizan en los conmutadores. Nótese que un inyector tiene la misma funcionalidad que un puerto de salida de un conmutador.

CAPÍTULO 5

Validación y Resultados

En este capítulo ofrecemos una evaluación del mecanismo de QoS integrado en la arquitectura PEAK. El objetivo de este capítulo es el de ofrecer una aproximación al comportamiento y al coste de los mecanismos integrados para ofrecer calidad de servicio.

En primer lugar ofrecemos varias simulaciones con un solo conmutador, demostrando las diferencias existentes al instanciar distintas configuraciones que utilizan árbitros diferentes. Las simulaciones se realizan sobre un conmutador que tiene soporte de dos VNs y cuatro VCs en cada VN. Por lo tanto, el sistema implementa 8 colas en la red. Se realizan tres simulaciones diferentes variando los árbitros de canales virtuales y los árbitros de puerto.

En segundo lugar ofrecemos dos simulaciones con el fin de demostrar las diferencias existentes al instanciar una red 2×2 de cuatro conmutadores con soporte de redes virtuales y canales virtuales frente a la configuración inicial sin soporte de QoS. Para provocar congestión en la red se genera tráfico desde dos orígenes distintos a un mismo destino. En estas pruebas se muestra un cálculo de la tasa de recepción en los módulos eyectores por lo que las diferencias se identifican fácilmente.

En tercer lugar ofrecemos casos de uso del mecanismo de canales virtuales, demostrando el funcionamiento correcto bajo las diferentes aplicaciones que se utilizan actualmente en PEAK. Estas aplicaciones son básicamente el arranque del sistema operativo PEAKos que se ejecuta sobre el sistema y una aplicación de multiplicación de matrices. Para ello, se ha implementado un conjunto básico de procesadores en PEAK, formando un sistema 4×2 de *tiles*. Cada procesador posee una caché de instrucciones de 1 KB, una caché de datos de 16 KB. La cache de segundo nivel está formada por 8 bancos de cache, cada uno de 64 KB. El sistema mejorado integra dos redes virtuales, donde cada red virtual integra cuatro canales virtuales. Por tanto, el sistema implementa 8 colas.

En cuarto lugar ofrecemos un estudio de prestaciones del sistema de QoS implementado. Para ello, hemos desarrollado un mecanismo de generación de trá-

fico sintético en cada procesador. Este mecanismo permite configurar cada procesador para que inyecte tráfico sintético a todos los destinos del sistema. En este caso hemos implementado también un sistema con 8 procesadores, distribuidos en 8 *tiles*, formando una configuración 4×2.

En quinto lugar, ofrecemos un estudio sobre el cumplimiento de la reserva de ancho de banda implementado en el árbitro del conmutador. Este estudio también se ha realizado sobre un sistema 4×2 donde varios inyectores intervienen generando diferentes flujos de tráfico a diferentes destinos de forma que atraviesan por un mismo enlace físico a lo largo de la red.

Por último, ofrecemos un estudio de costes de recursos para diferentes configuraciones del sistema QoS desarrollado. En concreto, analizamos los recursos utilizados para la implementación de diferentes configuraciones de VNs y VCs, poniendo el énfasis en los sobrecostes respecto al sistema base que no utiliza canales virtuales.

Un aspecto importante a tener en cuenta es que en aquellas configuraciones que implementan dos redes virtuales de cuatro canales virtuales cada una, los canales virtuales del 0 al 3 se usan en la VN0 y del 4 al 7 en la VN1.

5.1 Prueba con un Conmutador

El conmutador analizado posee cinco puertos de entrada y cinco puertos de salida implementando dos redes virtuales, cada una de ella con cuatro canales virtuales. Este conmutador tiene un *testbench* asociado que permite generar tráfico para probar el mecanismo de QoS implementado. En concreto, se ha inyectado tráfico por todos los puertos del conmutador indicando como destino de salida el puerto del Este. Los mensajes inyectados en cada puerto son tres mensajes de un solo *flit* de tamaño.

En el primer test se ha analizado el comportamiento del conmutador con el árbitro de asignación dinámica de canales virtuales sin reserva de ancho de banda para las VNs. La Figura 5.1 representa el resultado de la simulación en formas de onda.

Las primeras señales que se muestran son la de reloj y la de *reset*. Después se muestran las señales de entrada en la que en cada puerto se visualiza el contenido del *flit*, el tipo de ese *flit* y el bit de válido. Por último se muestran las señales del puerto al que van dirigidos todos los *flits* de este test.

Tal y como se observa en la figura anterior el árbitro de VCs asigna dinámicamente a cada paquete un canal de salida disponible en la misma VN y cuando el mensaje termina este canal virtual vuelve a estar disponible para ser asignado de nuevo. Nótese que la utilización del puerto de salida es máxima y todos los *flits* que son inyectados salen por el conmutador.

Otro aspecto a tener en cuenta es la latencia del conmutador desde que algún *flit* es inyectado por un puerto de entrada hasta que sale por algún puerto de salida. La latencia total del conmutador es de cuatro ciclos. El primer ciclo para el IB y el RT, el segundo para el VA, después SA y por último XOP.

El segundo test se ha implementado con el árbitro de asignación estática de VCs y sin reserva de ancho de banda para las VNs. En la Figura 5.2 se observa la forma en la que han quedado asignados los canales virtuales a cada mensaje. Se puede comprobar que a cada mensaje se le ha asignado el mismo canal de salida que el canal de entrada al que proviene. En este caso el enlace de salida también tiene una utilización máxima y todos los mensajes que entran salen por el puerto al que están destinados.

El tercer test realizado con un conmutador muestra el comportamiento de un árbitro con asignación dinámica de VCs pero con una reserva de ancho de banda para la VN0 del 20 % y para la VN1 del 80 %. En este test, los árbitros que asignan el puerto de salida contienen un vector de diez pesos en los cuales existen repartidos ocho con el valor 0 y dos con el valor 1. De esta forma cada vez que realizan peticiones al puerto de salida se aplican prioridades a una VN o a otra. En la Figura 5.3 se puede observar que se cumplen las prioridades en los primeros diez *flits* de salida, así como la utilización máxima del enlace. También se comprueba que el número de *flits* que se han inyectado es el mismo que los que se eyectan.

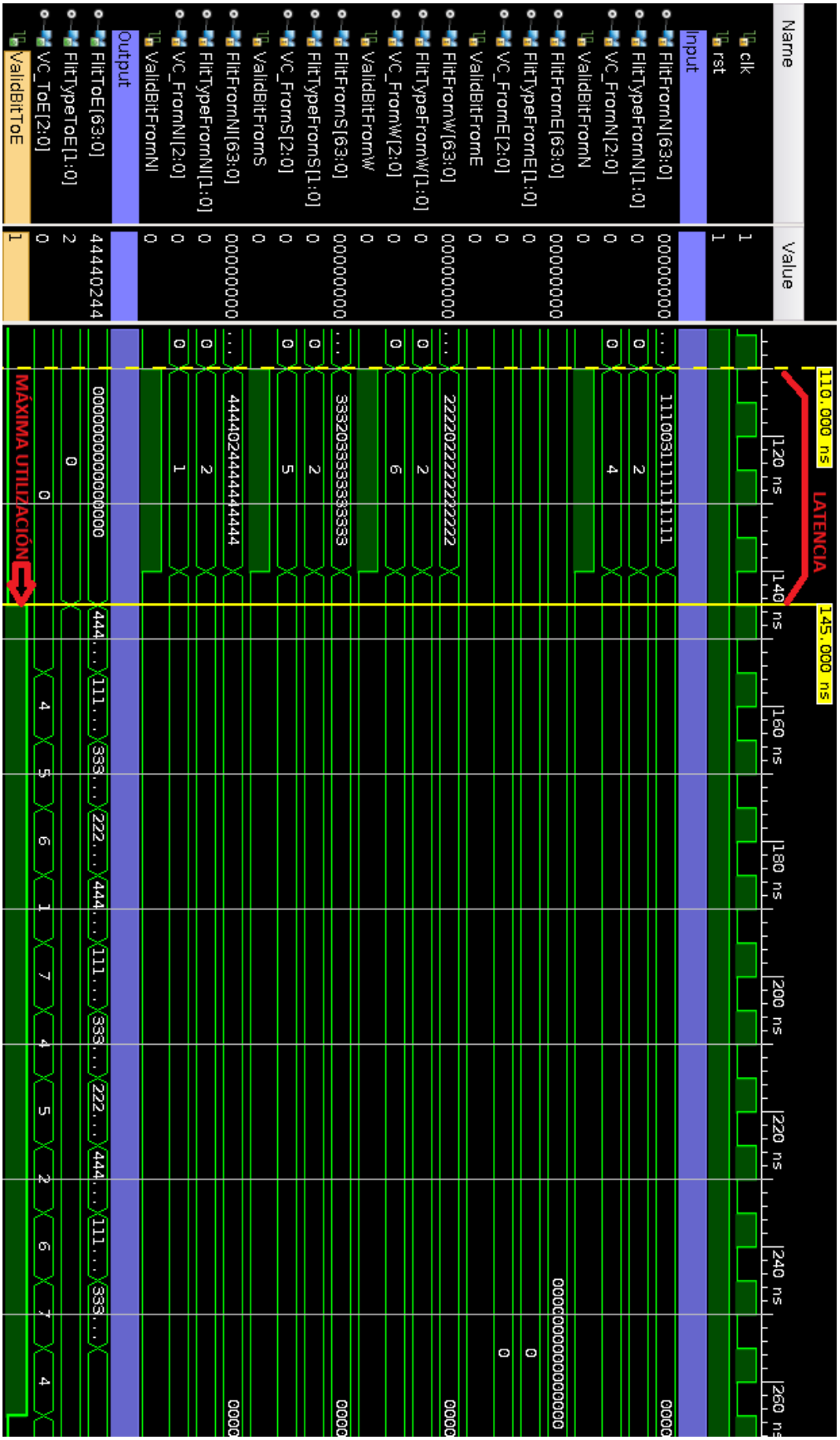

Figura 5.1: Prueba con un conmutador de 2 VNs de 4 VCs con árbitros de asignación dinámica

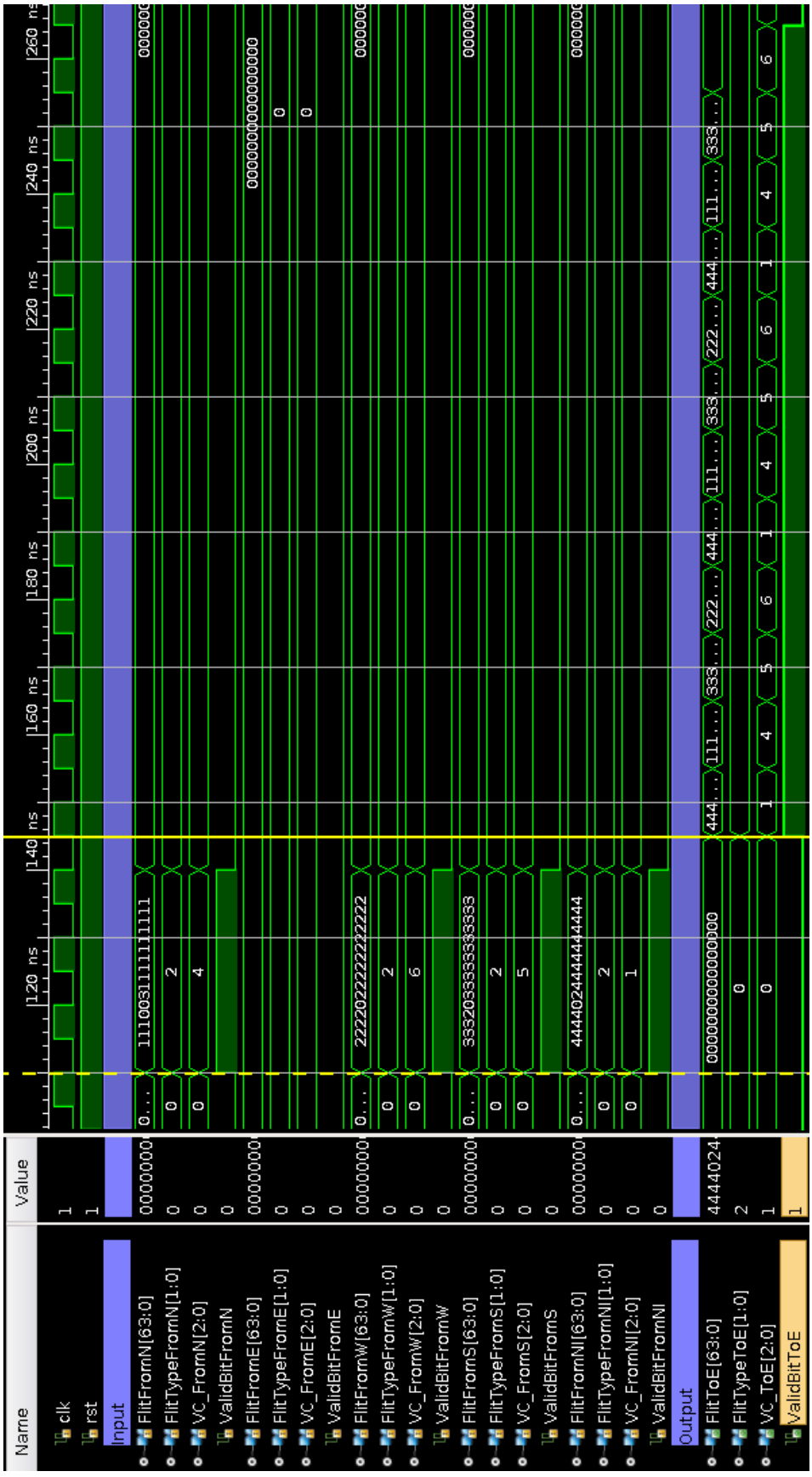

Figura 5.2: Prueba con un conmutador de 2 VNs de 4 VCs con árbitros de asignación estática

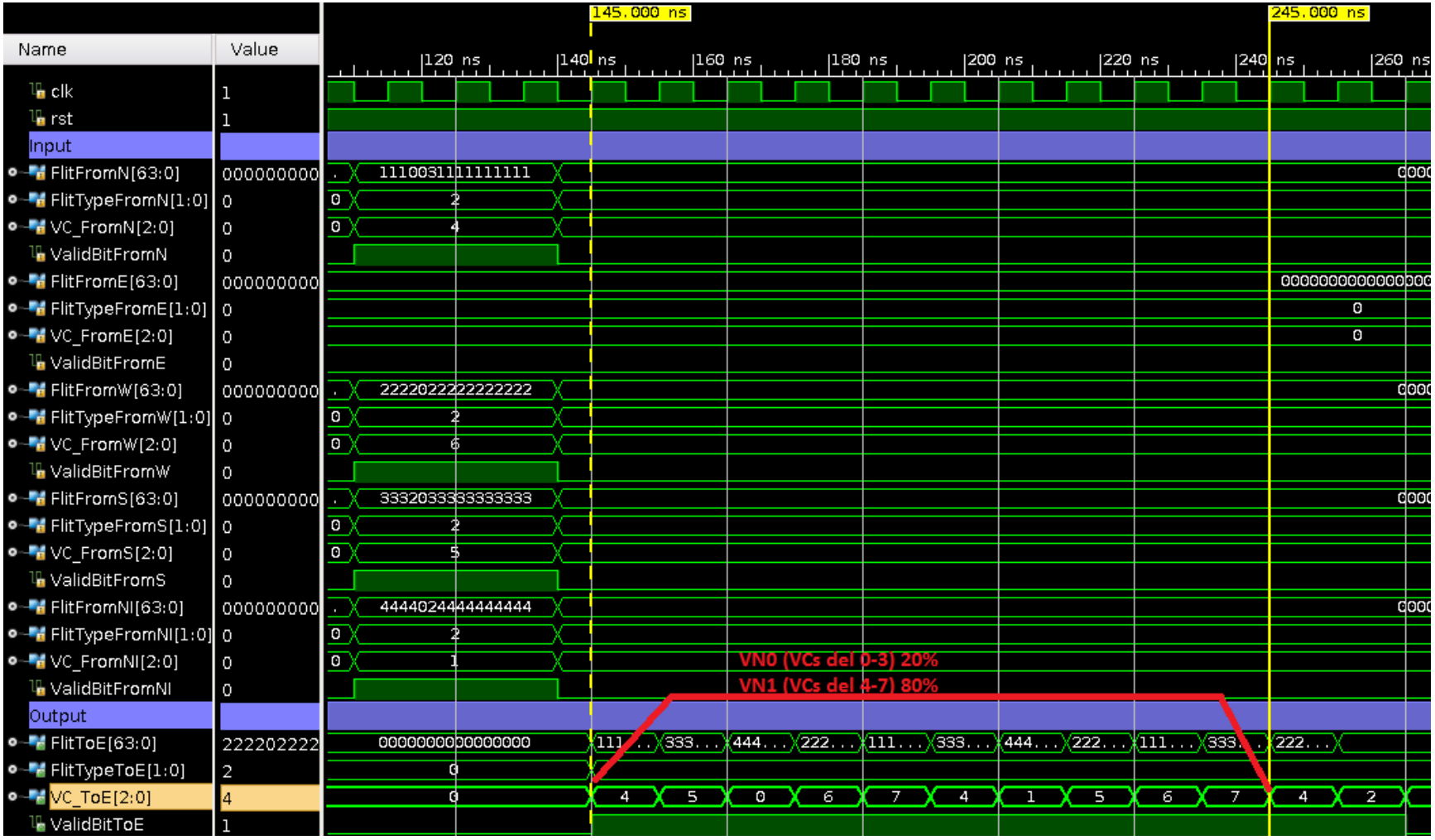

Figura 5.3: Prueba con un conmutador de 2 VNs de 4 VCs con árbitros de asignación dinámica y con pesos, 20 % para la VN0 y 80 % para la VN1

5.2 Prueba con una Red de 2 × 2

En esta segunda prueba la red contiene cuatro conmutadores con sus respectivos inyectores y eyectores en los puertos locales. Esta pequeña red nos permitirá ver las diferencias entre la red con soporte para VNs y VCs frente a la red base de PEAK. Se ha comparado la red con dos VNs con cuatro VCs cada una frente a la configuración base. Estas dos configuraciones implementan árbitros RR en los puertos de salida de cada conmutador.

Para evaluar los dos sistemas se ha definido un *testbench* que permite generar tráfico. En concreto, se inyecta tráfico desde los conmutadores 1 y 2 con destinos diferentes tal y como se muestra en la Figura 5.4. En todos los casos los mensajes tienen una longitud de cien *flits*, la cabecera, 98 de *payload* y la cola.

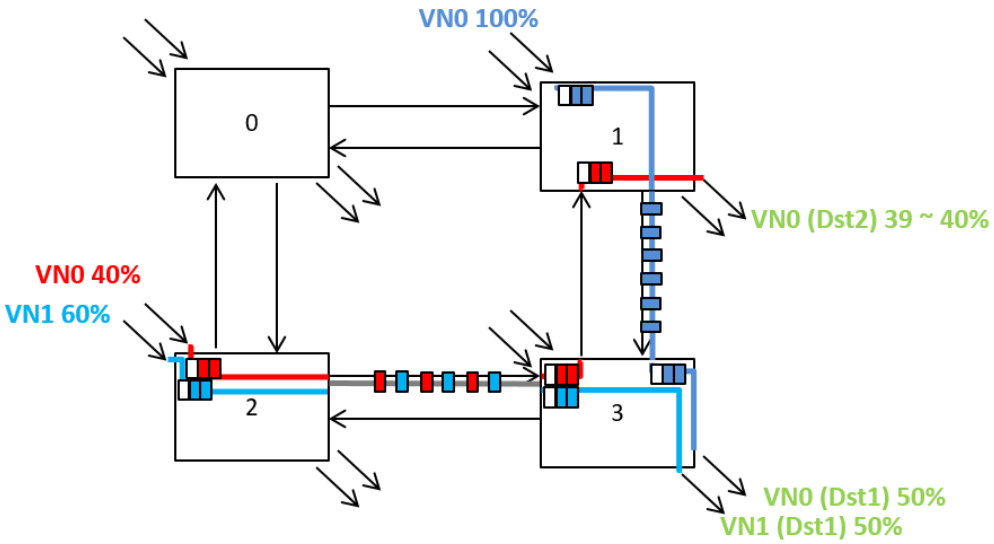

Figura 5.4: Prueba de una red 2 × 2 con conmutadores con 2 VNs de 4 VCs con árbitros de asignación dinámica

La tasa de inyección es del 100 % para el tráfico generado en el conmutador 1 y por la red virtual VN0. La tasa de inyección desde el conmutador 2 es del 40 % y 60 % para el tráfico generado para las redes VN0 y VN1, respectivamente. El conmutador 2 inyecta por VN0 hacia el destino 1, mientras que inyecta por VN1 hacia el destino 3.

Como se puede comprobar en la Figura 5.5 la configuración actual de la red es capaz de transportar todos los paquetes, encaminarlos y entregarlos a sus correspondientes destinos con una buena utilización de los puertos de salida. En la parte inferior de la figura se muestra la tasa de recepción real de los eyectores. En concreto, el nodo 3 recibe una tasa de tráfico del 100 %, repartida entre VN0 y VN1 con un 50 % por VN. El nodo 1 recibe la tasa del 40 % generada por el nodo 2. Por tanto, no existe ninguna ineficiencia en la transmisión de los mensajes.

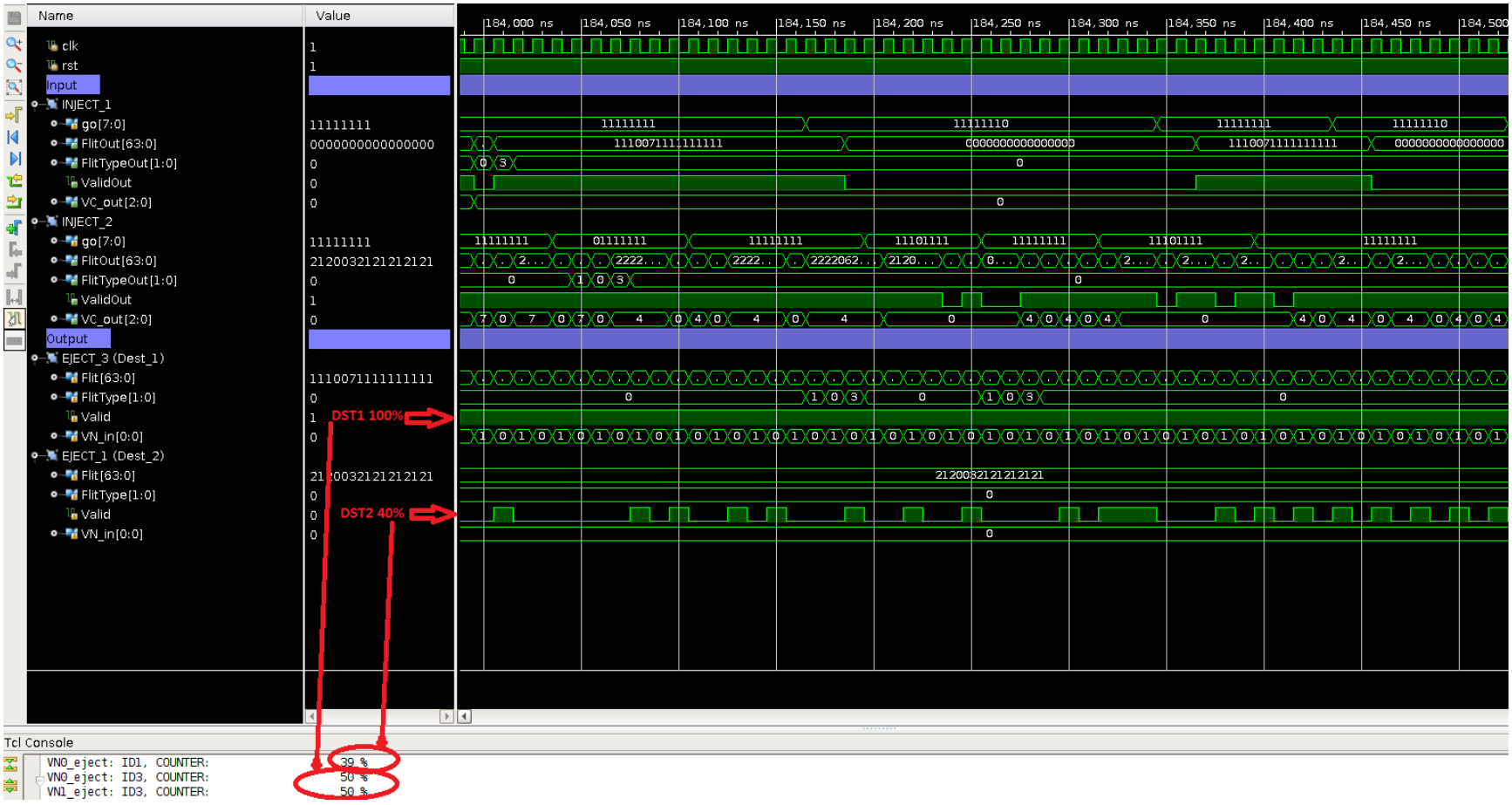

Figura 5.5: Prueba de una red 2 × 2 con conmutadores con 2 VNs de 4 VCs con árbitros de asignación dinámica

En la configuración base también se ha inyectado desde los mismos fuentes y a las mismas tasas de inyección. Pero en este caso, se puede observar en la Figura 5.6 cómo uno de los destinos no recibe a la misma tasa que se le está inyectando. Esto ocurre principalmente porque se genera contención entre los mensajes que atraviesan el enlace del conmutador 2 al 3 al no tener canales virtuales. Es por ello que únicamente existe una cola situada en cada puerto de entrada. Cuando un mensaje que va dirigido al destino 1 no puede avanzar, porque hay otro mensaje atravesando el mismo conmutador, y con el mismo destino se ejercita el control de flujo. El control de flujo detiene el flujo entrante, en este caso proveniente del conmutador 2 y es por ello que los mensajes dirigidos al destino 2 experimentan esta contención produciéndose una degradación en las prestaciones para el destino 2. Este fenómeno es conocido como *Head-of-line blocking*.

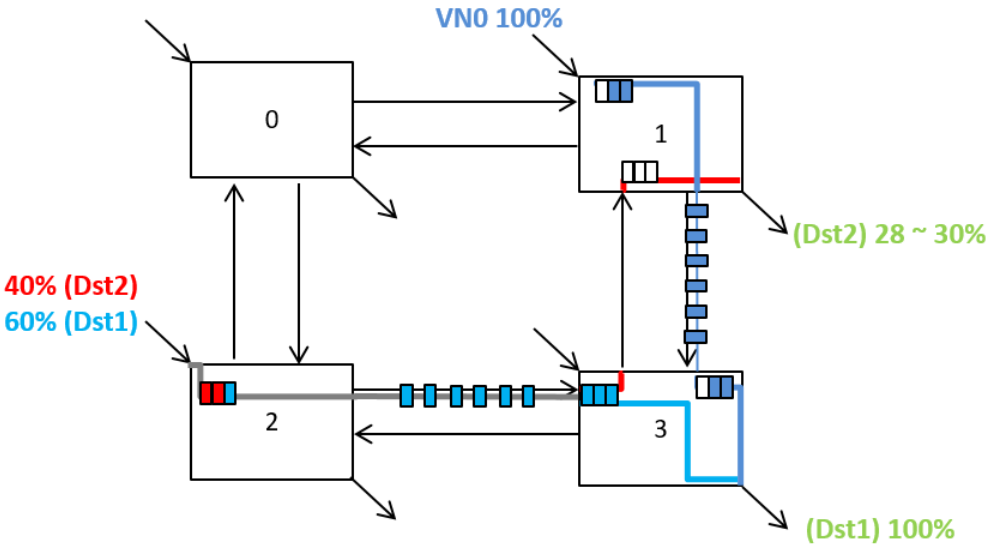

Figura 5.6: Prueba de una red 2 × 2 con conmutadores con la configuración base y árbitros de asignación dinámica

Como se puede comprobar en la Figura 5.7 la configuración base de la red no consigue una tasa para el destino 2 igual a la que se le está inyectando desde la fuente del conmutador 2. El eyector del destino 2 muestra una tasa del 30 % aproximadamente y podemos decir que experimenta una degradación de un 10 % de las prestaciones.

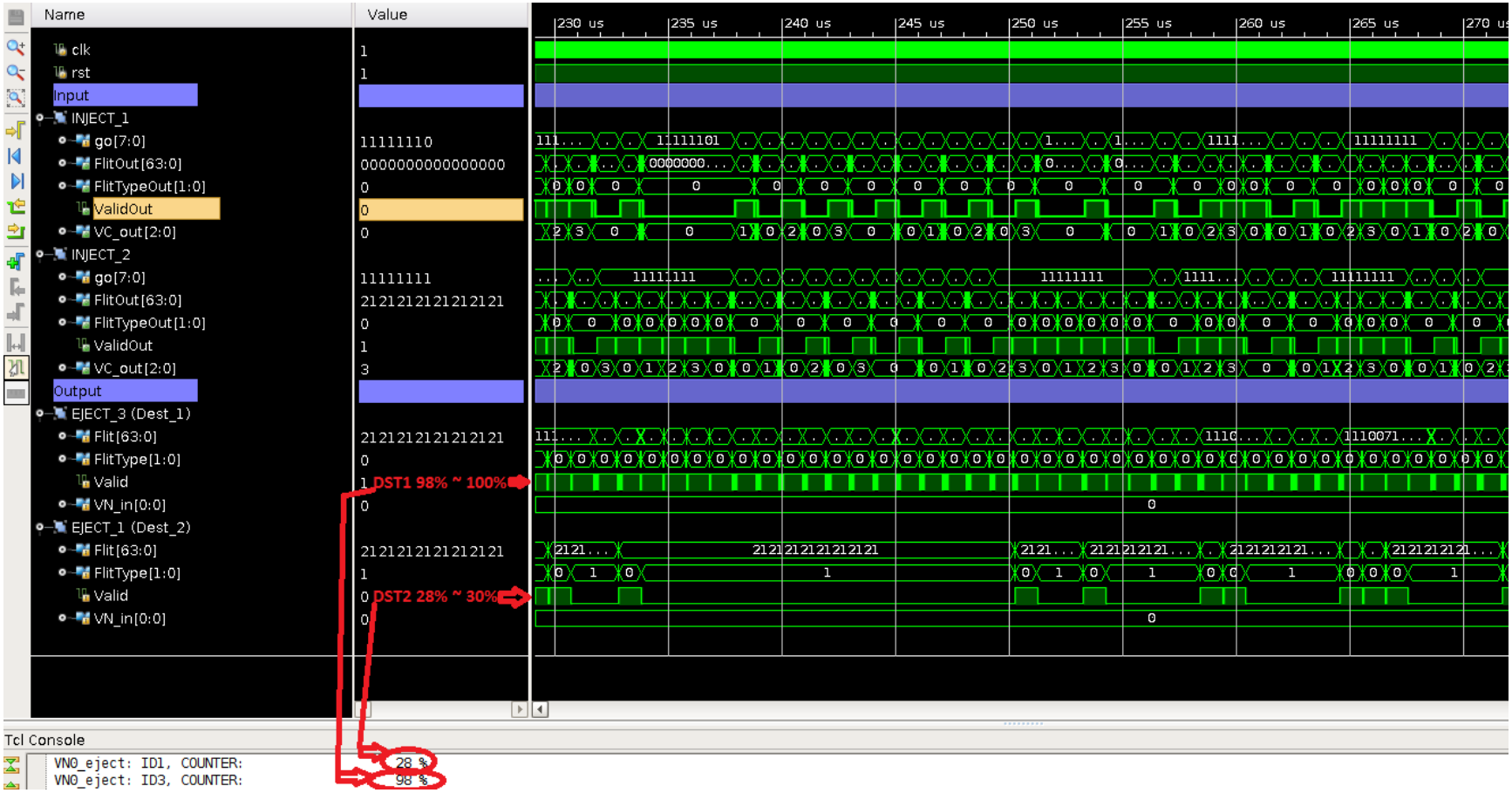

Figura 5.7: Señales de salida de la red 2 × 2 con la configuración base y árbitros de asignación dinámica

5.3 Prueba Real con 8 Procesadores en PEAK

Una vez hemos validado nuestro sistema con diferentes pruebas de simulación, procedemos a implementar todo el sistema y a realizar pruebas reales en PEAK. Cabe destacar que cada vez que realizamos el proceso de implementación las herramientas necesitan varias horas para generar el fichero de *bitfile* que posteriormente configura las placas FPGAs.

A continuación se muestra una ejecución real del sistema PEAK con la implementación de 8 *tiles* y la red con calidad de servicio. En la Figura 5.8 se observa como el sistema se inicia por medio del comando *boot*. Este comando hace un reset a todo el sistema, carga el sistema operativo PEAKos en una dirección de memoria concreta y luego carga el protocolo de coherencia de memorias caches. Seguidamente despierta al procesador con ID 0 puesto que es el *manager* y es el que recibe las interrupciones del sistema a través de los comandos por consola que le enviamos al sistema.

En el inicio del sistema se imprimen algunos mensajes por consola. Estos mensajes muestran la arquitectura que ha detectado el sistema, en número de procesadores detectados, el modelo de la placa FPGA en el que se implementa así como el tamaño de memoria que posee el sistema entre otras cosas.

Si enviamos el comando *show* se muestran las estadísticas del sistema desde el momento en que se ha realizado el último inicio.

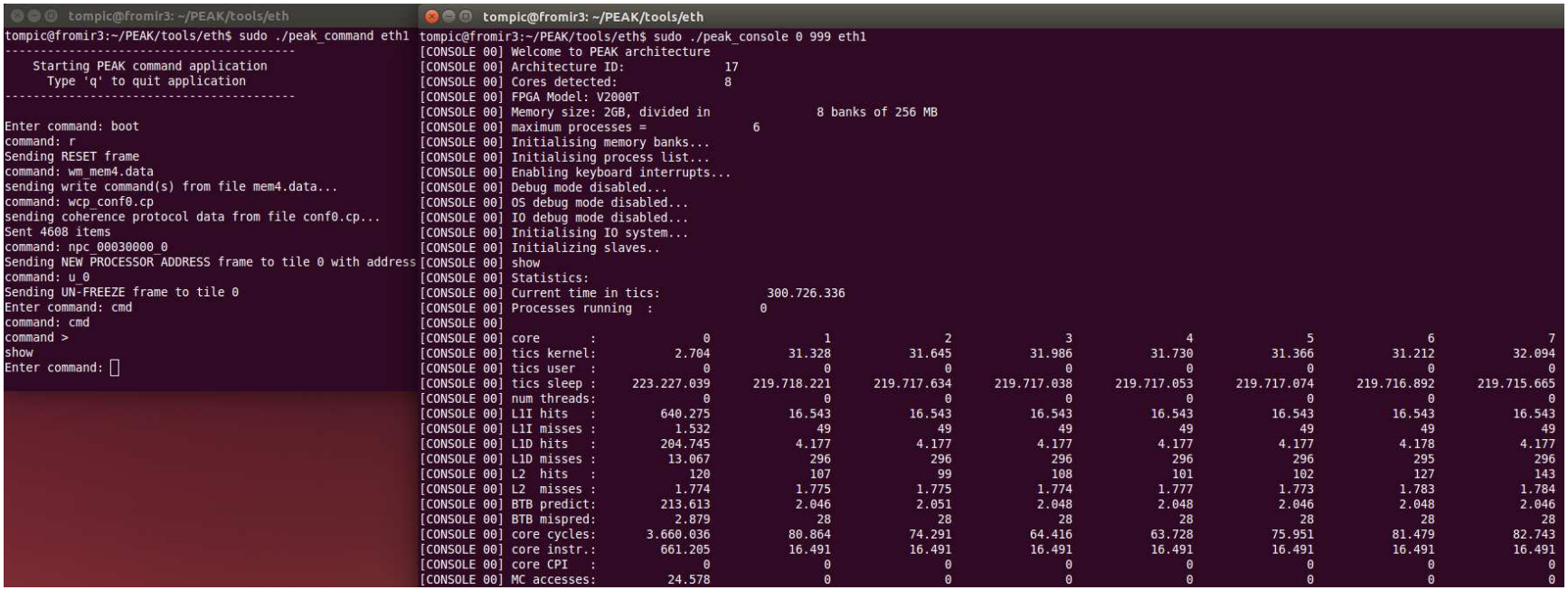

Figura 5.8: Ejecución real de sistema con una implementación de 8 procesadores en PEAK

5.4 Validación de Operatividad del Mecanismo en PEAK

Una vez se ha comprobado que el sistema arranca correctamente iniciamos diferentes pruebas para validar su correcto funcionamiento en situaciones de estrés. Para saturar el sistema hemos usado dos pruebas. La primera consiste en un programa de multiplicación de matrices que varía la magnitud de las matrices conforme se van resolviendo. La segunda consiste en un inyector de tráfico sintético que genera mensajes con destinos aleatorios y por lo tanto llena la red de mensajes.

En la Figura 5.9 se representa una ejecución real del programa de multiplicación de matrices.

Tal y como se muestra en la figura hay seis procesadores ejecutando multiplicaciones de matrices. Este programa genera múltiples accesos a las memorias caches por lo que si nos fijamos en la parte inferior de la figura, el comando *show* muestra estos accesos en las estadísticas.

Por otro lado, se ha diseñado un generador de tráfico sintético que nos ayuda a realizar pruebas exhaustivas de la red, sometiéndola a elevados niveles de tráfico. Ejecutando el comando *switch_inject* junto con los argumentos correspondientes (<origen><modo><tamaño del mensaje><tasa de inyección><VN><destino>) se activa el generador de tráfico sintético. Normalmente utilizamos el modo 2 para inyectar tráfico desde un origen a destinos aleatorios a través de las redes virtuales de datos. Un aspecto importante a tener en cuenta es que al tener destinos aleatorios cabe la posibilidad de que el destino coincida con el nodo inyector y en este caso no se inyectan mensajes por lo que se pierde tasa de inyección. También se utiliza el modo 4 para inyectar tráfico desde un nodo origen a un destino determinado y utilizando una VN determinada.

Por otro lado, el comando *nshow* muestra las estadísticas de las redes de interconexión de cada nodo. Se calcula tanto el número de *flits*, así como la productividad de cada enlace de la red. Por lo tanto, podemos apreciar la utilización de la red en cada ejecución. Cabe destacar que las estadísticas se pueden poner a cero y calcular a partir de la última puesta a cero. Para ello se usan los tics que se muestran en la parte de arriba de la captura. Si queremos saber la equivalencia en segundos solo tenemos que dividir *delta time* entre la velocidad a la que se ejecuta el sistema (en este caso 5 Mhz).

En la Figura 5.10 se representa una inyección en el Nodo 7 con destinos aleatorios. Nótese que en este caso se está inyectando a una tasa del 100 % lo que corresponde a una productividad de $\frac{1}{8nodos}$ repartida en dos VNs. Es lo mismo que decir un 6.25 % para cada destino y VN. Por lo tanto, la suma de la productividad en los enlaces Este y Norte del nodo inyector en las dos VNs corresponde a un 86 % y la pérdida de inyección total se estima un 12.5 %. En total todo suma

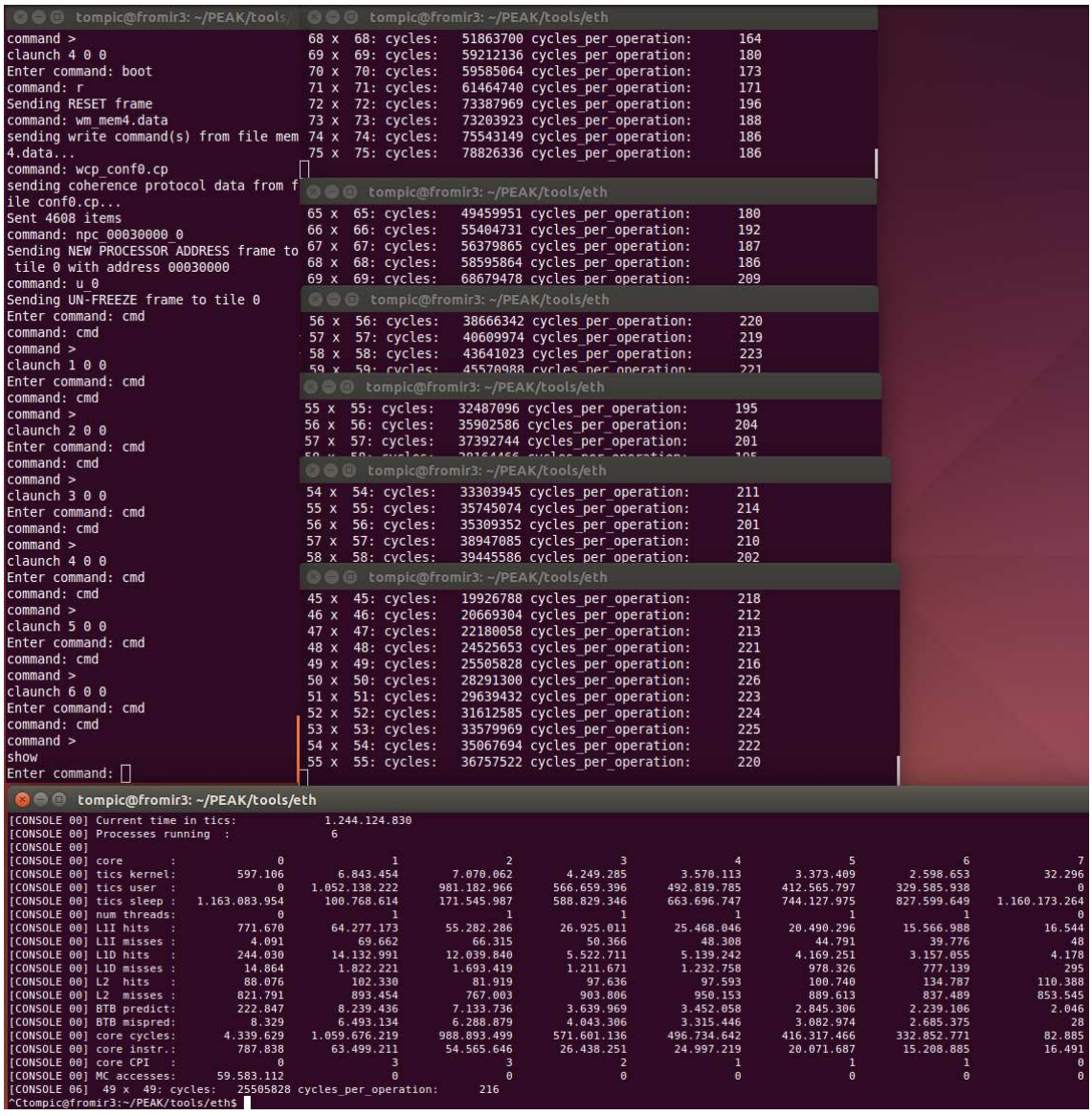

Figura 5.9: Prueba real con 8 procesadores en PEAK, de los cuales 6 ejecutando multiplicaciones de matrices

98 % que es aproximadamente la tasa de inyección. Este 2 % restante se debe al ciclo que se pierde entre la cola de un mensaje y la cabecera del siguiente mensaje en la etapa de encaminamiento.

De otro modo, si observamos la Figura 5.11 se muestra una ejecución en la que se inyectan desde todos los nodos hacia destinos aleatorios. Así pues experimentamos la inmensa cantidad de *flits* que es capaz de soportar la red en pocos segundos (en este caso $\frac{89\,196\,189}{5\,000\,000}$=17,84 segundos). Es interesante resaltar la limitación que se ve reflejada en este caso por los puertos locales. Esto se debe al ancho de banda de la bisección de la red.

Sabiendo que la red es una malla 2×4 en la bisección se hallan 4 enlaces unidireccionales. Aunque podemos inyectar desde 8 nodos y eyectar a 8 nodos la bisección es de 4 enlaces y por lo tanto la red en una situación de tráfico uniforme está limitada al 50 % (VN0 25 % y VN1 25 %). A pesar de la sobrecarga a la que se ha sometido la red, la red continua estable, el mecanismo de control de flujo actúa y no hay ninguna pérdida de mensajes.

5.5 Validación en Implementación de Reserva de Anchos de Banda

El objetivo principal es ofrecer calidad de servicio en la red de interconexión y en este apartado se ha realizado una prueba para validar que realmente se cumple la reserva de ancho de banda. Se ha implementado una red 2×4 con dos VNs de 4 VCs cada una y con árbitros en los puertos de salida con pesos. Para reservar el ancho de banda se define un vector con diez valores que determina la prioridad de salida en la técnica de arbitraje para cada puerto de salida. Este vector se define tal y como se ha descrito en la prueba con un conmutador en una de las variables globales del sistema. En concreto se han definido prioridades de un 20 % para la VN0 y de un 80 % para la VN1.

Para realizar esta prueba se ha inyectado un flujo de datos a lo largo de la red pensado con el fin de que se compartan enlaces intermedios entre dos flujos de datos y los árbitros actúen de acuerdo a las prioridades establecidas. En este caso se ha inyectado por la VN1 del nodo 0 a una tasa del 100 % con destino al nodo 7. Por la VN0 se ha inyectado desde los nodos 1 y 2 a una tasa del 100 % en ambos nodos y con destinos al nodo 3. Puesto que se usa un encaminamiento XY los tres flujos de datos atraviesan el puerto Este del nodo 2 y dos de ellos el puerto Este del nodo 1.

La salida por consola de estadísticas sobre la red en la ejecución real se muestra en la Figura 5.13 y el resumen de la situación se encuentra representado en el esquema de la Figura 5.12.

Según los resultados de ejecución se aprecia una reserva del ancho de banda para la VN1 de un 80 % a lo largo de toda la red. El enlace compartido entre el

```
[CONSOLE 00] Network statistics:
[CONSOLE 00] Current time:   1.402.658.084   Delta time:      88.032.517
[CONSOLE 00] Switch:              [          0          ]          [          1          ]          [          2          ]          [          3          ]
[CONSOLE 00] VN0
[CONSOLE 00] Flits local:       5.483.000 (          6%)       5.474.001 (          6%)       5.500.082 (          6%)       5.497.516 (          6%)
[CONSOLE 00] Flits south:               0 (          0%)              10 (          0%)              20 (          0%)              10 (          0%)
[CONSOLE 00] Flits west:                0 (          0%)               0 (          0%)               0 (          0%)               0 (          0%)
[CONSOLE 00] Flits east:               31 (          0%)              21 (          0%)              10 (          0%)               0 (          0%)
[CONSOLE 00] Flits north:               0 (          0%)               0 (          0%)               0 (          0%)               0 (          0%)
[CONSOLE 00] VN1
[CONSOLE 00] Flits local:       5.512.039 (          6%)       5.512.000 (          6%)       5.511.001 (          6%)       5.500.000 (          6%)
[CONSOLE 00] Flits south:               0 (          0%)               0 (          0%)               0 (          0%)               0 (          0%)
[CONSOLE 00] Flits west:                0 (          0%)               0 (          0%)               0 (          0%)               0 (          0%)
[CONSOLE 00] Flits east:                0 (          0%)               0 (          0%)               0 (          0%)               0 (          0%)
[CONSOLE 00] Flits north:               0 (          0%)               0 (          0%)               0 (          0%)               0 (          0%)
[CONSOLE 00] VN2
[CONSOLE 00] Flits local:             205 (          0%)              26 (          0%)              26 (          0%)              26 (          0%)
[CONSOLE 00] Flits south:              27 (          0%)              36 (          0%)              36 (          0%)              36 (          0%)
[CONSOLE 00] Flits west:                0 (          0%)              81 (          0%)              54 (          0%)              27 (          0%)
[CONSOLE 00] Flits east:              201 (          0%)             134 (          0%)              67 (          0%)               0 (          0%)
[CONSOLE 00] Flits north:               0 (          0%)               0 (          0%)               0 (          0%)               0 (          0%)

[CONSOLE 00] Switch:              [          4          ]          [          5          ]          [          6          ]          [          7          ]
[CONSOLE 00] VN0
[CONSOLE 00] Flits local:       5.507.001 (          6%)       5.493.020 (          6%)       5.504.020 (          6%)              12 (          0%)
[CONSOLE 00] Flits south:               0 (          0%)               0 (          0%)               0 (          0%)               0 (          0%)
[CONSOLE 00] Flits west:                0 (          0%)      10.994.221 (         12%)      21.969.285 (         24%)      32.980.009 (         37%)
[CONSOLE 00] Flits east:                0 (          0%)               1 (          0%)               3 (          0%)               0 (          0%)
[CONSOLE 00] Flits north:       5.489.000 (          6%)       5.481.001 (          6%)       5.504.434 (          6%)       5.507.337 (          6%)
[CONSOLE 00] VN1
[CONSOLE 00] Flits local:       5.494.000 (          6%)       5.518.000 (          6%)       5.511.000 (          6%)               1 (          0%)
[CONSOLE 00] Flits south:               0 (          0%)               0 (          0%)               0 (          0%)               0 (          0%)
[CONSOLE 00] Flits west:                0 (          0%)      11.013.039 (         12%)      22.048.029 (         25%)      33.078.957 (         37%)
[CONSOLE 00] Flits east:                0 (          0%)               1 (          0%)               1 (          0%)               0 (          0%)
[CONSOLE 00] Flits north:       5.520.039 (          6%)       5.513.000 (          6%)       5.517.001 (          6%)       5.505.000 (          6%)
[CONSOLE 00] VN2
[CONSOLE 00] Flits local:              26 (          0%)              35 (          0%)              35 (          0%)              35 (          0%)
[CONSOLE 00] Flits south:               0 (          0%)               0 (          0%)               0 (          0%)               0 (          0%)
[CONSOLE 00] Flits west:                0 (          0%)             111 (          0%)              74 (          0%)              37 (          0%)
[CONSOLE 00] Flits east:                0 (          0%)               0 (          0%)               0 (          0%)               0 (          0%)
[CONSOLE 00] Flits north:             146 (          0%)               0 (          0%)               0 (          0%)               0 (          0%)
```

Figura 5.10: Prueba real con inyección exhaustiva de paquetes a la red desde un solo origen

```
[CONSOLE 00] Network statistics:
[CONSOLE 00] Current time:   2.543.973.846   Delta time:      89.196.189
[CONSOLE 00] Switch:              [          0          ]          [          1          ]          [          2          ]          [          3          ]
[CONSOLE 00] VN0
[CONSOLE 00] Flits local:      23.009.877 (          25%)      22.986.568 (          25%)      22.943.743 (          25%)      22.715.000 (          25%)
[CONSOLE 00] Flits south:      13.091.000 (          14%)      13.046.631 (          14%)      13.106.000 (          14%)      13.138.000 (          14%)
[CONSOLE 00] Flits west:               0 (           0%)      19.611.849 (          21%)      26.271.604 (          29%)      19.917.000 (          22%)
[CONSOLE 00] Flits east:       19.739.177 (          22%)      26.052.000 (          29%)      19.558.009 (          21%)               0 (           0%)
[CONSOLE 00] Flits north:              0 (           0%)               0 (           0%)               0 (           0%)               0 (           0%)
[CONSOLE 00] VN1
[CONSOLE 00] Flits local:      22.827.668 (          25%)      22.946.143 (          25%)      22.931.645 (          25%)      22.906.000 (          25%)
[CONSOLE 00] Flits south:      13.136.059 (          14%)      13.143.144 (          14%)      13.188.504 (          14%)      13.251.389 (          14%)
[CONSOLE 00] Flits west:               0 (           0%)      19.608.640 (          21%)      26.229.634 (          29%)      19.686.586 (          22%)
[CONSOLE 00] Flits east:       19.813.524 (          22%)      26.476.475 (          29%)      19.609.048 (          21%)               0 (           0%)
[CONSOLE 00] Flits north:              0 (           0%)               0 (           0%)               0 (           0%)               0 (           0%)
[CONSOLE 00] VN2
[CONSOLE 00] Flits local:            175 (           0%)              26 (           0%)              26 (           0%)              26 (           0%)
[CONSOLE 00] Flits south:             27 (           0%)              27 (           0%)              27 (           0%)              27 (           0%)
[CONSOLE 00] Flits west:               0 (           0%)              81 (           0%)              54 (           0%)              27 (           0%)
[CONSOLE 00] Flits east:             174 (           0%)             116 (           0%)              58 (           0%)               0 (           0%)
[CONSOLE 00] Flits north:              0 (           0%)               0 (           0%)               0 (           0%)               0 (           0%)


                                  [          4          ]          [          5          ]          [          6          ]          [          7          ]
CONSOLE 00] Switch:
CONSOLE 00] VN0                22.841.319 (          25%)      22.884.548 (          25%)      22.925.924 (          25%)      23.017.000 (          25%)
CONSOLE 00] Flits local:               0 (           0%)               0 (           0%)               0 (           0%)               0 (           0%)
CONSOLE 00] Flits south:               0 (           0%)      19.570.994 (          21%)      26.122.022 (          29%)      19.612.015 (          21%)
CONSOLE 00] Flits west:        19.848.440 (          22%)      26.123.000 (          29%)      19.623.000 (          21%)               0 (           0%)
CONSOLE 00] Flits east:        13.140.000 (          14%)      13.119.049 (          14%)      13.115.000 (          14%)      13.074.000 (          14%)
CONSOLE 00] Flits north:
CONSOLE 00] VN1                23.050.157 (          25%)      22.970.514 (          25%)      23.111.000 (          25%)      23.050.217 (          25%)
CONSOLE 00] Flits local:               0 (           0%)               0 (           0%)               0 (           0%)               0 (           0%)
CONSOLE 00] Flits south:               0 (           0%)      19.723.107 (          22%)      26.312.022 (          29%)      19.896.000 (          22%)
CONSOLE 00] Flits west:        19.840.532 (          22%)      26.319.379 (          29%)      19.667.000 (          22%)               0 (           0%)
CONSOLE 00] Flits east:        13.064.000 (          14%)      13.175.155 (          14%)      13.012.000 (          14%)      13.185.000 (          14%)
CONSOLE 00] Flits north:
CONSOLE 00] VN2                       26 (           0%)              26 (           0%)              26 (           0%)              26 (           0%)
CONSOLE 00] Flits local:               0 (           0%)               0 (           0%)               0 (           0%)               0 (           0%)
CONSOLE 00] Flits south:               0 (           0%)              81 (           0%)              54 (           0%)              27 (           0%)
CONSOLE 00] Flits west:                0 (           0%)               0 (           0%)               0 (           0%)               0 (           0%)
CONSOLE 00] Flits east:              116 (           0%)               0 (           0%)               0 (           0%)               0 (           0%)
CONSOLE 00] Flits north:
```

Figura 5.11: Prueba real con inyección exhaustiva de paquetes a la red desde varios orígenes

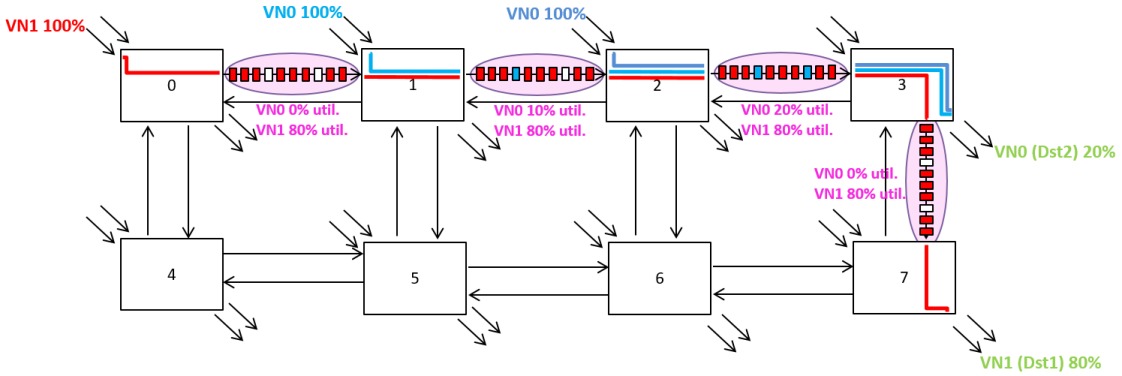

Figura 5.12: Resumen de la situación para validar una implementación de reserva de anchos de banda en una red de interconexión 2×4

nodo 2 y el nodo 3 tiene una utilización total del 100 % aplicando los criterios de prioridades correctamente. Nótese que la utilización en la VN0 se ve repartida equitativamente entre los nodos 1 y 2 conforme se representa en la utilización del enlace entre el nodo 1 y 2. El control de flujo actúa y reduce la tasa de inyección del nodo 1 y 2 únicamente al 10 % para cada uno de estos flujos porque el árbitro situado en el puerto Este del nodo 2 aplica un 20 % para la VN1. En consecuencia a lo anterior, los paquetes que se entregan por la VN0 en nodo 3 resultan un 20 % y en el nodo 7 hasta el 80 %.

Como conclusión a esta prueba podemos decir que en el supuesto de lanzar una aplicación con un ámbito de criticidad alto podríamos reservarle los recursos de la red virtual VN1 exclusivamente. Las otras aplicaciones del sistema podrían utilizar la VN0 y el sistema funcionaría reservando un ancho de banda total del 80 % para la aplicación crítica.

```
[CONSOLE 00] Network statistics:
[CONSOLE 00] Current time:     899.781.604   Delta time:      14.112.528
[CONSOLE 00] Switch:            [          0       ]       [          1       ]       [          2       ]       [          3       ]
[CONSOLE 00] VN0
[CONSOLE 00] Flits local:              0 (          0%)            0 (          0%)            0 (          0%)        2.926.111 (         20%)
[CONSOLE 00] Flits south:              0 (          0%)            0 (          0%)            0 (          0%)                0 (          0%)
[CONSOLE 00] Flits west:               0 (          0%)            0 (          0%)            0 (          0%)                0 (          0%)
[CONSOLE 00] Flits east:               0 (          0%)    1.469.745 (         10%)    2.940.157 (         20%)                0 (          0%)
[CONSOLE 00] Flits north:              0 (          0%)            0 (          0%)            0 (          0%)                0 (          0%)
[CONSOLE 00] VN1
[CONSOLE 00] Flits local:              0 (          0%)            0 (          0%)            0 (          0%)                0 (          0%)
[CONSOLE 00] Flits south:              0 (          0%)            0 (          0%)            0 (          0%)       11.328.663 (         80%)
[CONSOLE 00] Flits west:               0 (          0%)            0 (          0%)            0 (          0%)                0 (          0%)
[CONSOLE 00] Flits east:      11.359.433 (         80%)   11.361.795 (         80%)   11.364.154 (         80%)                0 (          0%)
[CONSOLE 00] Flits north:              0 (          0%)            0 (          0%)            0 (          0%)                0 (          0%)
[CONSOLE 00] VN2
[CONSOLE 00] Flits local:             70 (          0%)           11 (          0%)           11 (          0%)               11 (          0%)
[CONSOLE 00] Flits south:             12 (          0%)           12 (          0%)           12 (          0%)               12 (          0%)
[CONSOLE 00] Flits west:               0 (          0%)           36 (          0%)           24 (          0%)               12 (          0%)
[CONSOLE 00] Flits east:              84 (          0%)           56 (          0%)           28 (          0%)                0 (          0%)
[CONSOLE 00] Flits north:              0 (          0%)            0 (          0%)            0 (          0%)                0 (          0%)


CONSOLE 00] Switch:             [          4       ]       [          5       ]       [          6       ]       [          7       ]
CONSOLE 00] VN0
CONSOLE 00] Flits local:               0 (          0%)            0 (          0%)            0 (          0%)                0 (          0%)
CONSOLE 00] Flits south:               0 (          0%)            0 (          0%)            0 (          0%)                0 (          0%)
CONSOLE 00] Flits west:                0 (          0%)            0 (          0%)            0 (          0%)                0 (          0%)
CONSOLE 00] Flits east:                0 (          0%)            0 (          0%)            0 (          0%)                0 (          0%)
CONSOLE 00] Flits north:               0 (          0%)            0 (          0%)            0 (          0%)                0 (          0%)
CONSOLE 00] VN1
CONSOLE 00] Flits local:               0 (          0%)            0 (          0%)            0 (          0%)       11.318.605 (         80%)
CONSOLE 00] Flits south:               0 (          0%)            0 (          0%)            0 (          0%)                0 (          0%)
CONSOLE 00] Flits west:                0 (          0%)            0 (          0%)            0 (          0%)                0 (          0%)
CONSOLE 00] Flits east:                0 (          0%)            0 (          0%)            0 (          0%)                0 (          0%)
CONSOLE 00] Flits north:               0 (          0%)            0 (          0%)            0 (          0%)                0 (          0%)
CONSOLE 00] VN2
CONSOLE 00] Flits local:              11 (          0%)           11 (          0%)           11 (          0%)               11 (          0%)
CONSOLE 00] Flits south:               0 (          0%)            0 (          0%)            0 (          0%)                0 (          0%)
CONSOLE 00] Flits west:                0 (          0%)           36 (          0%)           24 (          0%)               12 (          0%)
CONSOLE 00] Flits east:                0 (          0%)            0 (          0%)            0 (          0%)                0 (          0%)
CONSOLE 00] Flits north:              56 (          0%)            0 (          0%)            0 (          0%)                0 (          0%)
```

Figura 5.13: Resultados de la implementación de reserva de anchos de banda en una red de interconexión 2×4

5.6 Costes de Implementación

En esta última sección realizamos un estudio de la ocupación en recursos de cada una de las versiones de conmutadores diseñadas. Este estudio lo realizamos en términos de recursos necesarios para la implementación en la FPGA. Representamos en forma de tablas y gráficas los datos sobre la síntesis de las diferentes configuraciones del conmutador con su correspondiente relación en cuanto a uso de recursos internos en la FPGA.

La cantidad de recursos lógicos necesarios se encuentra expresada en tablas LUT (*Lookup Table*). Dichas unidades son tablas de verdad que implementan pequeñas funciones lógicas, y al interconectarse entre sí, implementan la funcionalidad completa del módulo. Por otro lado, las partes de memoria necesaria para soportar los bloques alojados se encuentran representadas en registros.

En la Tabla 5.1 se pueden observar el número de configuraciones que se han establecido para la red de interconexión. Estas se diferencian por el número de redes virtuales, número de canales virtuales implementados en cada VN o los árbitros de VCs y de los puertos de salida.

Conf.	**VNs**	**VCs**	**Árbitro VCs**	**Árbitro Pto. Salida**
Baseline	-	-	-	RR
2_1_S_RR	2	1	Estático	RR
2_2_S_RR	2	2	Estático	RR
2_4_S_RR	2	4	Estático	RR
2_4_D_RR	2	4	Dinámico	RR
2_4_D_20_80	2	4	Dinámico	Pesos(20/80)

Tabla 5.1: Diferentes componentes que incluye cada configuración de la red

La configuración *Baseline* no dispone de soporte de redes virtuales ni canales virtuales. La configuración *2_1_S_RR* es la que podría sustituir directamente a las dos redes físicas (de configuración base) con las que partía el sistema PEAK porque implementa dos redes con un solo canal virtual en cada una. Finalmente, las configuraciones con mas recursos *2_4_D_RR* y *2_4_D_20_80* se pueden usar para ámbitos en los que la red de interconexión tenga una carga muy elevada. La primera tiene un reparto de carga equitativo entre sus dos VNs debido a que implementa árbitros dinámicos para VCs y otros *round-robin* en los puertos de salida. La segunda implementa un árbitro con prioridades del 20 % para la VN0 y de un 80 % para la VN1.

La Tabla 5.2 muestra la cantidad de recursos de ocupación por cada conmutador que implementa la red. Estos recursos están calculados por las herramientas de síntesis teniendo en cuenta una FPGA Virtex 7 2000tflg1925-1. Tal y como se puede observar en la tabla, las cantidades referentes a número de tablas LUT y re-

gistros se encuentran normalizadas frente a la configuración base. De este modo se ve claramente como aumentan los recursos utilizados según se implementan configuraciones mas complejas. Estas cantidades ascienden hasta un 11,55 en la lógica de la configuración con mas coste y un 7,01 en cuanto a recursos de memoria. Los datos de la tabla se reflejan gráficamente en las Figura 5.14 que representa el porcentaje de recursos en cuanto a lógica se refiere y la Figura 5.15 referente a los recursos de memoria.

Conf.	Tablas LUT	%Tablas LUT	Regs	%Regs
Baseline	1	0,14 %	1	0,03 %
2_1_S_RR	2,36	0,32 %	1,85	0,05 %
2_2_S_RR	4,62	0,63 %	3,52	0,10 %
2_4_S_RR	11,20	1,53 %	6,88	0,19 %
2_4_D_RR	13,55	1,85 %	7,01	0,19 %
2_4_D_20_80	11,55	1,57 %	7,01	0,19 %

Tabla 5.2: Resultados de síntesis del conmutador en cada configuración (LUTs normalizados frente a la configuración *Baseline*)

Como se puede apreciar en la Figura 5.14, conforme aumentan el número de canales virtuales aumentan las tablas LUT utilizadas. Esto se debe al aumento de cableado entre los módulos del conmutador, aumento de lógica de encaminamiento y mas lógica a causa de las peticiones que llegan a los árbitros. Nótese la diferencias que existen entre las configuraciones con cuatro canales virtuales. La implementación con árbitros de VCs estáticos necesita menos lógica que la que implementa árbitros dinámicos. Si nos fijamos en los recursos utilizados en la configuración con pesos frente a la *2_4_D_RR* también es menor. Podemos concluir que los árbitros dinámicos de VCs y los árbitros *round-robin* en los puertos de salida utilizan mucha mas lógica que los árbitros estáticos o con pesos.

Con otra perspectiva, la Figura 5.15 representa gráficamente los registros utilizados de la FPGA por un conmutador de la red que implementa cada configuración. En esta gráfica se reflejan claramente la cantidad de colas utilizadas en los puertos de entrada. Las colas de entrada se implementan como recursos de memoria.

En la configuración base se implementa una cola por cada puerto de entrada y la utilización de recursos de memoria es muy baja. La implementación *2_1_S_RR* utiliza dos colas en cada puerto de entrada porque incluye dos redes virtuales y cada VN tiene un canal virtual. Es por ello que utiliza el doble de recursos frente a la *Baseline*. Sin embargo, hay que tener presente que ne la configuración *Baseline* se necesitan 2 conmutadores, por lo que la diferencia de recursos se cancela. Además, si nos fijamos en las tres implementaciones de cuatro canales virtuales, todas ellas utilizan prácticamente la misma cantidad de registros de memoria porque implementan las mismas colas en cada puerto de entrada.

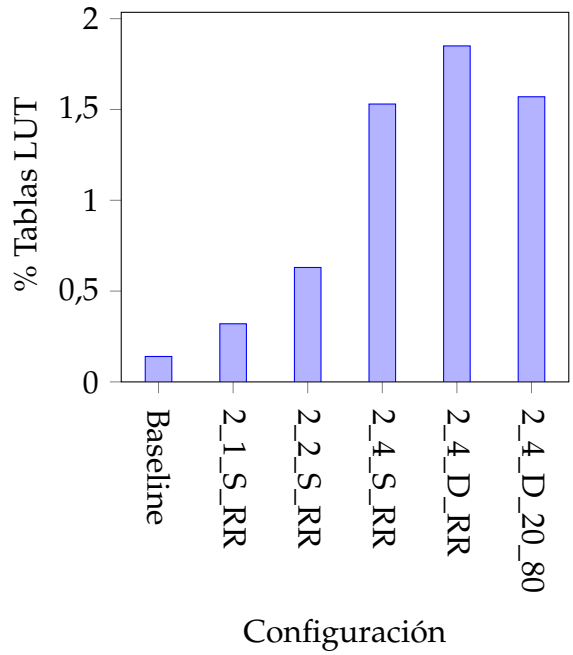

Figura 5.14: Gráfica con los resultados de síntesis en tablas *LUT* de todas las configuraciones

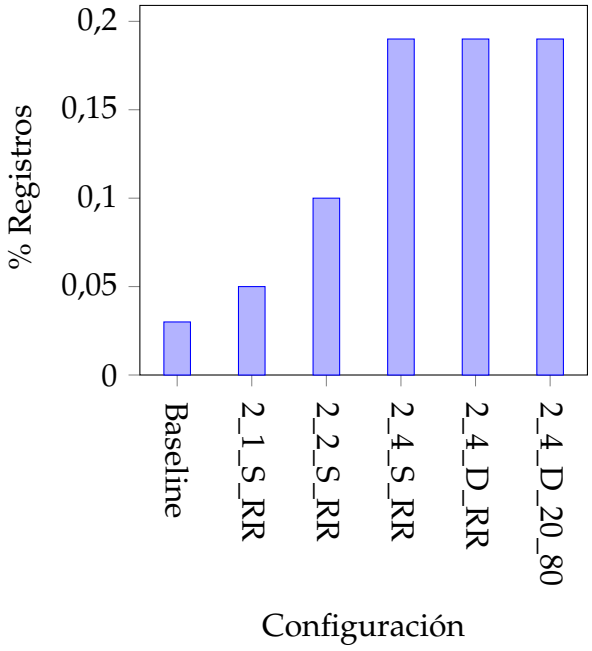

Figura 5.15: Resultados de síntesis en registros en cada configuración del conmutador

Como conclusión, podemos decir que un sistema con soporte para QoS tiene un sobrecoste marginal cuando el número de colas es el mismo que el caso ba-

se sin QoS. Además, este coste crece de forma lineal con el número de canales virtuales que queramos añadir.

CAPÍTULO 6

Conclusiones

En este capítulo se expone a modo de resumen qué objetivos se han visto alcanzados durante el desarrollo del trabajo, así como las posibles ampliaciones y trabajo futuro.

6.1 Consideraciones Finales

En este trabajo se puede decir que se han conseguido todos los objetivos propuestos con creces. Su implementación se ha abordado desde el inicio del máster hasta la penúltima semana ultimando detalles y realizando algunas correcciones.

Tal y como se ha demostrado en el Capítulo 5, el sistema cumple con las expectativas y los anchos de banda definidos se respetan para cada red virtual. Gracias a este trabajo, el proyecto MANGO incorporará la red de interconexión con soporte de calidad de servicio con capacidad de reservar anchos de banda en distintos flujos de datos.

Ahora la red se encuentra preparada para abordar grandes cantidades de tráfico si se implementan las configuraciones mas complejas, lo que la capacita para soportar sistemas con cargas muy altas de trafico a través de la red como puedan ser las aplicaciones de HPC de MANGO mencionadas al inicio de esta memoria.

También se han abierto las puertas a futuros estudios sobre el comportamiento de distintas aplicaciones ejecutándose de forma paralela en este el sistema.

6.2 Ampliaciones y Trabajo Futuro

Como posible mejora, se podría añadir una función en el sistema operativo PEAKos que permita, del mismo modo que se envían los comandos de inyección de tráfico a través de un registro en el *TILEREG*, poder modificar el vector

de prioridades una vez se tiene el sistema implementado en las FPGAs y con el sistema operativo iniciado.

En aras a mejorar la utilización de recursos se puede analizar la lógica que genera cada módulo del sistema en la red de interconexión y modificar el código con el fin de utilizar los menos recursos posibles. Esta optimización se realizará para subir la frecuencia de funcionamiento de PEAK.

Por último, puede resultar muy interesante llevar a cabo estudios sobre calidad de servicio y observar los comportamientos, resultados de ejecución o algunas limitaciones de nuestro sistema al ejecutar *benchmarks*. Todo ello nos daría lugar a nuevas publicaciones y aportaría ideas para mejorar el sistema.

Bibliografía

[1] J. Flich, D. Bertozzi, *Designing network on-chip architectures in the nanoscale era*, CRC Press, 2010

[2] J. Duato, S. Yalamanchili, L. Ni, *Interconnection Networks, an Engineering Approach*, IEEE Computer Society Press, 1997

[3] W. J. Dally, H. Aoki, *Deadlock -free adaptive routing in multicomputer networks using virtual channels*, IEEE Trans. on Parallel and Distributed Systems pp. 466-475, April, 1993

[4] J. W. Lee, A. Ng, and K. Asanovic, *Globally-synchronized frames for guaranteed quality-of-service in on-chip networks* in Proc. Int: Symp. Comput. Architecture, 2008, pp. 89–100

[5] E. F. Corrêa, L. A. P. Silva, F. R. Wagner, L. Carro, *Fitting the Router Characteristics in NoCs to Meet QoS Requirements*, in Proceedings of 20th SBCCI, ACM Press, 2007. pp. 105-110

[6] E. Rijpkema, K. Goossens, A. Radulescu, J. Dielissen, J. van Meerbergen, P. Wielage, and E. Waterlander. *Trade offs in the design of a router with both guaranteed and best-effort services for networks on chip*. in DATE, 2003

[7] L. F. Leung and C. Y. Tsui, *Optimal link scheduling on improving best-effort and guaranteed services performance in network-on-chip systems*, in Proc. Des. Autom. Conf., Jul. 2006, pp. 833–838

[8] K. Goossens et al., *A design flow for application-specific networks on chip with guaranteed performance to accelerate SoC design and verification*, in Proc. Des., Autom. Test Eur. Conf., Mar. 2005, pp. 1182–1187

[9] M. Millberg, E. Nilsson, R. Thid, and A. Jantsch, *Guaranteed bandwidth using looped containers in temporally disjoint networks within the Nostrum network on chip*, in Proc. Des., Autom. Test Eur. Conf., Feb. 2004, pp. 890–895

[10] Azeez Sanusi, Nan Wang and Magdy A. Bayoumi *Guaranteeing QoS with the Pipelined Multi-Channel Central Caching NoC Communication Architecture* pp75-78

[11] B. Grot, S. W. Keckler, and O. Mutlu, *Preemptive virtual clock: A flexible, efficient, and cost-effective QOS scheme for networks-onchip*, in IEEE/ACM Int. Symp. Microarchit. (MICRO), New York, 2009, pp. 163–174

[12] B. Grot et al.Kilo-NoC: *a heterogeneous network-on-chip architecture for scalability and service guarantees*. International Symposium on Computer Architecture, 39(3):401–412, 2011

[13] J. Flich and J. Duato, *Logic based distributed routing for NOCs*, IEEE computer architecture letters, vol. 7, no. 1, January-June 2008