Driver para Multicore Heterogéneo con Procesador de Petri

Orlando Micolini[#1], Agustín Martina [#2], Alfonso Maschio[#3], Eschoyez Maxiniliano[#4]

[#] *Laboratorio de Arquitectura de Computadoras, Universidad Nacional de Córdoba,*
Facultad de Ciencias Exactas Físicas y Naturales.
Av.Velez Sarsfield 299. Córdoba, Argentina.

[1] Omicolini@compuar.com

[2] sagostino@Gmail.com

[3] maschio.Alfonso@Gmail.com

[4]eschoyez.univ@gmail.com

***Abstract*— This paper expose a technique that simplified the connection between a multicore system that has an IP Core and a Cortex A9, with an operated system.**

This particular application use a Pretis Net Processor, developed in the Computer Architecture Laboratory of the FCEFyN-UNC, holding the ability of the processor to solve concurrency problems, simplifying the execution of parallel code like a disconnected sequential systems and avoiding the introduction of new instructions to the architecture. Furthermore, this shows the performance of the architecture, proposing improvements

The Driver has been accomplish, supporting the abstraction of the architecture functionalities and serving the features to the user in a simple and efficient way.

***Resumen*— Este trabajo presenta una técnica que simplifica la conexión entre un sistema multicore compuesto por un IP Core y un Cortex A9, con un sistema operativo.**

Como caso de aplicación, se utilizó un Procesador de Redes de Petri, desarrollado en el Laboratorio de Arquitectura de Computadoras de la FCEFyN-UNC, manteniendo la capacidad del procesador para resolver problemas de concurrencia, simplificando la ejecución paralela como sistemas secuenciales desacoplados y no agregando nuevas instrucciones a la arquitectura. Además, se ha determinado las prestaciones de la arquitectura, proponiendo mejoras.

Se logró un driver que abstrae las funcionalidades de la arquitectura y las expone al usuario de forma simple y eficiente.

Keywords— IP Core, Driver, Peti Net, Sistema Operativo.

I. INTRODUCCIÓN

La tecnología actual nos permite tener un procesador discreto multicore con una FPGA y un Sistema Operativo (SO) en un mismo componente integrado (conjunto Procesador-FPGA-SO). Los IP Core nos permiten reutilizar la FPGA como un componente de hardware. La amplia gama de IP Cores disponibles, es uno de los factores por los cuales se puede mejorar la producción de soluciones. Como consecuencia hay un notable incremento de este tipo de soluciones.

Para diseñar e implementar sistemas de gran porte, donde son necesarios procesadores, IP Cores y sistemas operativos, es complicada la integración del IP Core en forma manual. Es decir, la programación "ad hoc" de cada requerimiento requiere recursos muy especializados en todas estas áreas. Nuestra propuesta es realizar un driver con capacidad de soportar distintos IP Cores para ser integrados a sistemas con multiprocesadores y sistemas operativos.

En el Laboratorio de Arquitectura de Computadoras, FCEFyN-UNC, se ha desarrollado un Procesador de Redes de Petri implementado como un IP Core [1]. Este soluciona problemas de concurrencia y además permite, en un programa, desacoplar la parte paralela de la serial, sin la necesidad de modificar el ISE (Patterson XX).

Con el objetivo de utilizar este procesador, se diseñó un driver que integra el IP Core al SO, manteniendo las características de ambos y potenciando al sistema operativo en cuanto a las prestaciones que permite un uso desacoplado del IP Core. Este driver permite el uso de diferentes. IP Core con transferencia de caracteres

Finalmente se analizan nuevas características que pueden ofrecer el conjunto PP.

En el segundo apartado se exponen los objetivos del trabajo, tanto generales como específicos. Después se desarrolla el marco teórico donde se referencian temas como Drivers de Dispositivos en Linux, Sistemas Embebidos, Buses de comunicación, Componentes y FPGA. En el cuarto apartado se aborda el estudio del problema, donde, además se selecciona una placa para implementación y evaluación la solución. Seguidamente se evalúan aspectos de la placa como s distintas de a distintos niveles de memoria, latencias de la placa y SO. En el punto VI se propone una solución. Finalmente se exponen los resultados y se presentan las conclusiones del trabajo.

II. OBJETIVOS

A. *Objetivo General*

Diseñar un controlador (módulo driver) que integre un IP Core, del PP, al conjunto Procesador-FPGA-SO, con el objetivo de abstraer las funcionalidades.

B. *Objetivos Específicos*

- Evaluar, seleccionar y validar las prestaciones de los sistemas CPU-FPGA de última tecnología disponible en el mercado actual.
- Determinar el mínimo conjunto de funcionalidades (primitivas) comunes para controlar el IP Core PP

con el fin de sistematizar la interacción entre la CPU, y generalizar a distintos IP Cores con el SO.
- Medir y comparar los mecanismos de concurrencia del SO con los del PP.
- Implementar las primitivas en el módulo driver que permitan la transparencia de utilización por el usuario.
- Validar la suficiencia del controlador para integrar el PP con el SO.

III. Marco Teórico

A. *Introducción.*

Hasta el momento, el PP sólo se había implementado con un procesador soft-core Micro-Blaze, corriendo un Xilkernel [1]. En este trabajo hemos implementado el PP con un procesador hard-core, en una placa de desarrollo ZedBoard, corriendo un SO robusto.

Si bien es posible realizar esta implementación con cualquier SO que soporte la arquitectura del procesador hard-core, hemos elegido el SO GNU/Linux. Seguidamente exponemos una breve introducción del diseño de los drivers a los fines de establecer los componentes necesarios para el desarrollo de nuestro driver. También es necesario medir los tiempos para determinar el desempeño del driver.

A continuación exponemos un breve estudio de los Sistemas Embebidos y los System-on-Chip con los fines de seleccionar la placa de desarrollo donde implementamos nuestra solución.

B. *Linux Device Drivers*

Los drivers [2] (controladores) poseen un rol especial dentro del kernel de Linux, son "cajas negras" que permiten al usuario ocultar los detalles de cómo funciona el dispositivo. El usuario puede interactuar utilizando un conjunto de llamadas estandarizadas, que son independientes del driver específico, lo que puede verse en la Fig. 1.

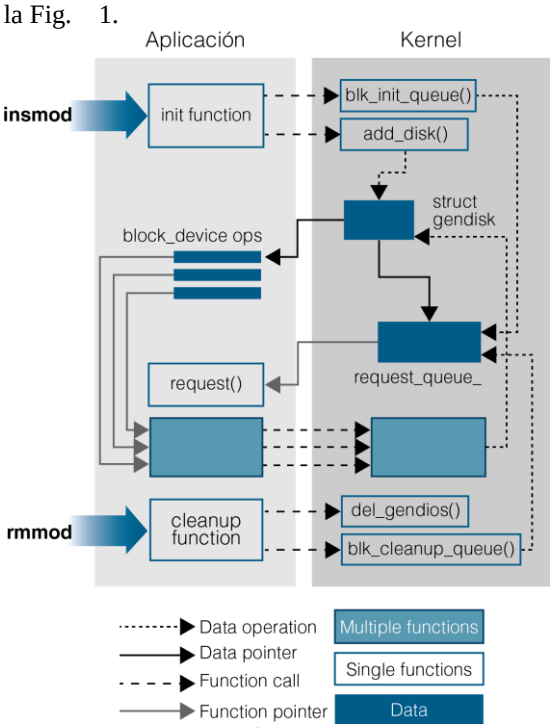

Fig. 1: **Flujo de información en un ambiente kernel-controlador**

El papel fundamental de estos es mapear estas llamadas a operaciones específicas propias del dispositivo, que actúan directamente sobre el hardware. Esto permite que los drivers sean diseñados separados del resto del kernel e instanciarlos en tiempo de ejecución cuando sean necesarios. Esta capacidad hace que los módulos de Linux sean fáciles de escribir e incluir.

El rol del módulo driver es proveer los mecanismos, no las políticas. La diferencia entre los mecanismos y las políticas es uno de los conceptos más importantes detrás del diseño de Unix. La mayoría de los problemas de programación se pueden dividir en dos partes:

1) El mecanismo: "Qué capacidades proporciona"
2) Las políticas: "Cómo pueden utilizarse estas capacidades"

Los sistemas Unix distinguen entre tres tipos de dispositivos. Estos son:

- Dispositivos de caracteres.
- Dispositivos de bloques.
- Interfaz de red.

El PP se enmarca entre los Dispositivos de Caracteres. Esto es porque puede ser accedido como un flujo de bytes.

Si bien esta clasificación denota tres categorías; también podemos agrupar los dos primeros en una misma. Se puede demostrar que un controlador de caracteres puede ser implementado como un controlador de bloque y viceversa. Los dispositivos de red poseen la característica de trabajar con eventos asíncronos y de tamaño variable.

Las funciones esenciales se dividen en dos grupos:

- Funciones de usuario
- Funciones de Kernel.

Como en general los distintos IP Cores requieren que distintos procesos puedan realizar lecturas y escrituras en memoria. Por esto, se debe mantener la capacidad que brinda el SO para encapsular procesos, es decir, áreas protegidas de memoria.

C. *Medición de tiempo en el driver*

El kernel de Linux exporta herramientas para corroborar el tiempo de ejecución de una tarea. Si bien son suficientes en variedad carecen de precisión. Para alcanzar la precisión requerida en este proyecto y disminuir la incertidumbre de medición es necesario recurrir a un mecanismo de medición de hardware. Por lo que recurrimos a los registros del procesador, donde se encuentran de manera exacta la cantidad de ciclos entre dos instantes de tiempo [3]. Este registro es accesible en modo privilegiado y requerirá un controlador para su consulta.

D. *Sistemas Embebidos*

Se entiende que un sistema embebido, es una combinación de componentes de Hardware y Software implementados en circuitos integrados o en lógica reconfigurable, diseñados para realizar una función o control específico dentro de un sistema que no es una computadora de uso general. Es por esto último que adquiere su nombre de embebido [4, 5].

Sus principales características son:

- El hardware que utiliza posee compontes en la misma placa y pocos puertos de expansión.
- Realiza funciones específicas.

- Poseen una mayor rigurosidad con respecto a sus requerimientos y fiabilidad que un sistema de uso general.

En particular se utilizó un Sistem-on-Chip (SoC), que es un subtipo de sistema embebido. Estos sistemas consisten en al menos un procesador integrado (una CPU), un controlador de memoria y un controlador de entrada y salida dentro de un único circuito integrado.

E. Comunicaciones on-Chip

Las redes de interconexión en el chip se implementan usando buses. El poder diseñar buses estandarizados permite que el remplazo y reutilización de módulos se simplifique. Los buses han adquirido peso propio en la eficiencia del sistema, dando paso a una gran cantidad de arquitecturas, topologías, algoritmos de acceso y diseños, generalmente, asociados a una arquitectura de procesador en particular.

Entre los buses más utilizados en la industria de los sistemas embebidos se encuentran el bus AMBA [6] y el PCI-Express [7].

F. Consideraciones en el diseño de sistemas embebidos

Las características computacionales de un sistema embebido difieren a las de un sistema de propósito general. Los sistemas embebidos diseñan y programan para un fin específico, en cambio, los de propósito general permiten ser reprogramados por el usuario cuantas veces se necesite. Es por esto que en los sistemas embebidos el diseño del software y del hardware se realiza en paralelo. Esto permite a los diseñadores tener la posibilidad de optimizar la aplicación para el hardware sobre el que se está diseñando, como memoria, consumo energético y requerimientos de tiempo real. Esta ventaja se pierde si se hace un diseño con técnicas de propósito general [3] [8].

El implementar un algoritmo en lógica programable permite explotar el paralelismo de los sistemas combinacionales y secuenciales.

Por último, las implementaciones en lógica programable pueden diseñarse para no ser interrumpidas, por lo que es posible obtener mejores latencias que con sistemas de propósito general, en igualdad de tecnología.

G. Programación por componentes

La programación orientada por componentes (POC)[9] permite construir programas con componentes de software pre-realizados que contienen código reutilizable. Estos componentes tienen ciertos estándares definidos, incluyendo interfaces, conexiones, versión, etc.

La POC se basa en las interfaces y en la composición. En este sentido, se puede decir que la POC es una programación orientada a las interfaces. Se entiende por interface lo que especifica el servicio que un cliente requiere de un componente. Los clientes de un componente no necesitan ningún conocimiento de cómo se implementa la interfaz.

Si bien existen muchos paradigmas y técnicas de programación, ninguno es tan cercano al concepto del hardware como la POC. En la atapa del diseño de hardware se definen los componentes y sus interfaces (Entradas y Salidas). Luego, en la etapa de implementación, el resultado son cajas negras (IP Cores) que poseen todas las características de un componente.

Extendemos la definición de componentes con una aproximación que nos facilita el diseño del software y del hardware en paralelo y su interconexión en sistemas embebidos. Esta aproximación implica que el hardware y el software sean componentes que se interconectan entre ellos. Aquí debemos ser cuidadosos en la definición de las interfaces. La Fig. 2, grafica esta aproximación.

Nuestra aproximación es abstraer al software como si fuese el hardware y usarlo como un componente de hardware.

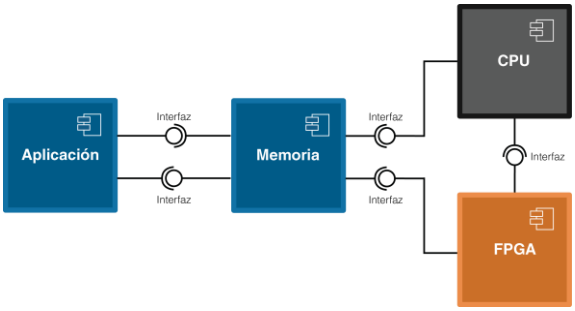

Fig. 2. Aplicaciones, Hardware y FPGA como componentes

H. FPGA

Los Field Programmable Gate Arrays, son circuitos integrados formados por bloques lógicos programables e interconectables entre sí. Permiten implementar sistemas de alta complejidad y flexibilidad, pudiendo combinarse con periféricos y microprocesadores; para formar los llamados «Sistemas programables en chip».

Otra alternativa, es hacer uso de núcleos de procesadores implementados haciendo uso de la lógica de la FPGA (Softcores), teniendo en cuenta que los bloques programados pueden ejecutarse concurrentemente.

Los bloques de lógica programable se denominan IP-Cores y los trataremos como si fuesen componentes de un paradigma POC.

IV. Estudio Del Problema

El problema se centra en encontrar la forma de poder generar una instancia del PP en un sistema embebido que posea un procesador con una arquitectura multicore y una FPGA en la misma pastilla, cumpliendo el requerimiento de que se encuentre a una misma distancia que la memoria caché L1[5] (poner referencia a paper de baldoni romano).

Como se propone dar soporte a sistemas reactivos y estos cuentan con gran demanda de sincronización, es necesario que el tiempo de decisión de una sincronización sea del orden del tiempo de procesamiento de una instrucción, que es el mejor desempeño que se puede esperar del sistema.

Por esto último se realizó un estudio sobre las distintas placas de desarrollo que posean un conjunto Procesador-FPGA-SO. Puesto que inferimos que al estar en la misma pastilla, el procesador y la FPGA, el retardo de comunicación entre ambos será menor que en otras alternativas

Como resultado de este estudio se seleccionó la placa de desarrollo ZedBoard que posee un chip Zynq de Xilinx y que posee características de nuestro interés, las que son:

- Procesador Cortex A9 dual core conectado por un bus AMBA 3 a una FPGA de Xilinx Atrix 7 con 85k de Celdas Lógicas, sobre el mismo *die*.
- 512mb de Ram.
- Soporta GNU/Linux

Para que cualquier usuario pueda ver al PP, lo tenemos que integrar al SO a través de un driver.

V. Sobre las Distintas Latencias de la Placa y SO

A. *Introducción*

Lamentablemente no se cuenta con una documentación referente a las latencias de la placa elegida, ya que el hardware es un prototipo de ingeniería en proceso de diseñado, por la comunidad. En particular, nos interesaban las latencias de acceso a los distintos niveles de memoria. Estos datos son necesarios a los fines de comparar las latencias que posee el SO en resolver problemas de concurrencia (sincronización) con respecto al PP.

Con el objetivo de obtener estos valores, realizamos mediciones de los ciclos de reloj que le demora al procesador acceder a cada nivel de memoria mediante un driver diseñado a este efecto. Los tiempos han sido medidos para los siguientes niveles de memoria y sus respectivas direcciones:

- Memoria on-Chip - OCM (0x00000000)
- Memoria DDR (0x00100000)
- FPGA (0x62600000).

B. *Mediciones Realizadas Sobre la Placa Sin SO*

Los resultados obtenidos se muestran en la Tabla I. En ella se observa que la OCM es la que se encuentra más cerca al procesador, con medias de 24,1 y 59,96 ciclos de reloj para lectura y escritura, respectivamente. La OCM posee la misma latencia que la memoria chache L2 [3].

TABLA I
DISTANCIAS, EN PULSOS, DESDE EL PROCESADOR A LOS DISTINTOS NIVELES DE MEMORIA Y LA RELACIÓN ENTRE ELLOS

	Lectura	**Escritura**
OMC	24,1	59,96
DDR	32	149
FPGA	131	149
FPGA/DDR	4,6	1
FPGA/OCM	5,41	2,52

Otro aspecto interesante que se puede observar es que si bien la memoria DDR se encuentra fuera del *die*, a cantidad de pulsos de reloj, ésta se encuentra más cerca que la FPGA, ya que es un 12,7% más rápida en lectura, pero es igual en escritura. Atribuimos este efecto a que la FPGA funciona a un sexto de la velocidad del Procesador.

C. *Mediciones Realizadas con SO*

A los fines de comparar los tiempos que le llevan al SO utilizar su API de sincronización, se midieron los tiempos que le lleva a GNU/Linux resolver un semáforo (un Wait y un Notify) tanto en monocore como en multicore. La TABLA II, muestra los resultados obtenidos.

TABLA II
TEMPOS, EN CICLOS, EN QUE DEMORA RESOLVER UN SEMÁFORO

	Una Corrida	**Mil Corridas**	**Relación**
Multicore	8.378	3.742	2,23
Monocore	8.378	1.817	5.61

Se evidencia la diferencia de ejecutar un problema de exclusión mutua en uno y dos cores. Cuando ejecutamos en un mismo core, existe un cambio de contexto, como beneficio la cache del core permite una rápida solución en la sincronización. Por otro lado, al ejecutarlo en cores distintos, tenemos el problema de que es imposible mantener en la caché L1 los semáforos e indefectiblemente se deben bloquear el bus para poder consultar el verdadero valor del semáforo. Esto se realiza bloqueando el bus entre L1 y L2, y es por esto que notamos un claro aumento en la latencia.

En el escenario de un procesador monocore el semáforo se aloja en la L1 , mientras que en multicore, el código y los datos del semáforo están presentes en la L2. Por lo que, la relación entre correr estos procesos en multicore y monocore varía entre 1,7 y 2,0 veces.

Corroboramos entonces que, si deseamos implementar un IP Core que resuelva problemas relacionados con la sincronización, este debe encontrarse a la distancia de la cache L2, tal como fue planteado por (referencia Baldoni). En este caso, puesto que debemos implementarlo en la FPGA de la placa seleccionada, es ahí donde se instancia. A los fines de evaluar el driver, se debe realizar una extrapolación que permita inferir la mejora del sistema si este fuese implementado con los tiempos de acceso a L2.

VI. Solución Propuesta

Se propone crear un controlador para GNU/Linux el cual permita facilitar desde el espacio de usuario el envío de señales desde y hacia el PP instanciado en la FPGA.

El PP posee doce registros de escritura y doce de lectura [10, 11], como este driver se ha generalizado cualquier IP Core, con transferencia de datos por caracteres. Puesto que el bus AMBA define como únicas primitivas la lectura y escritura, dividimos nuestros controladores en dos categorías.

- Driver de lectura.
- Driver de escritura.

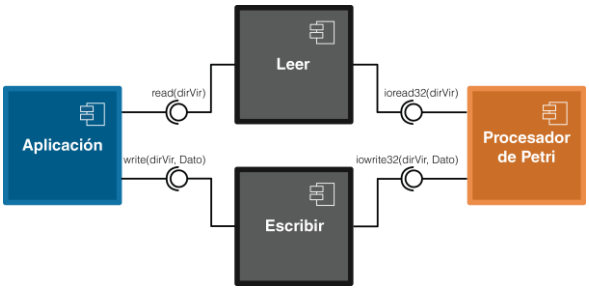

Fig. 3: Arquitectura de componentes del sistema propuesto

La implementación de diferentes controladores dentro de una misma categoría tiene impacto lógico, pero no físico. Los controladores dentro de una misma categoría se diferencian sólo por la dirección con la que interactúan, pero no en como lo hacen. El impacto de escribir una u otra

dirección está relacionado con la semántica de la interface del componente IP Core.

Se desarrollaron tantos controladores como registros, ya que el PP soluciona problemas de sincronización multicore. Si nuestro controlador fuese único, indefectiblemente debería poseer mecanismos de sincronización para sus secciones críticas. Esto entorpecería nuestras mediciones ya que un mecanismo de sincronización dependería del otro. Otro aspecto importante es el de las funciones paralelas Puesto que este diseño es para una arquitectura multicore, nuestro controlador debe ser capaz de sacar provecho de la arquitectura paralela.

TABLA III
MEDIDAS DE LECTURAS Y ESCRITURAS A LA FPGA CON Y SIN SO EN CICLOS DE RELOJ

	Con S.O.	**Sin S.O.**
Lectura de FPGA	1.829	131
Escritura de FPGA	1.552	149

VII. RESULTADOS OBTENIDOS

El objetivo de esta sección es la de evaluar el desempeño del uso del PP con respecto a la API del SO.

En primer lugar se realizó un estudio del tiempo que toma en realizar una Escritura y una Lectura al PP. Esto ayuda a entender la lejanía a la que se encuentra el mismo (*Tabla III*).

Al observar que los valores de lectura a la FPGA resultaron catorce veces superiores a los obtenidos sin SO y que los de escritora son diez veces superiores, procedimos a realizar la medición del driver sin acceso a la FPGA, con los fines de medir el overhead que agrega el driver a la lectura en el SO. Luego de haber realizado mil iteraciones se obtuvo una media de 862,16 ciclos de reloj, con un desvío de 8,56 ciclos. Que es el mínimo impacto (overhead) que agrega instanciar un módulo en el sistema.

A. Mediciones en Exclusión Mutua y Sincronización

Se realizó un estudio sobre distintos procesos en ciclos de sincronización y exclusión mutua, resueltos con Red de Petri, corriendo en un procesador monocore y en dualcore. El objetivo de este estudio fue el comparar con los valores obtenidos en la resolución de sincronización y exclusión mutua utilizando la API de GNU/LINUX.

Se observa en la *Tabla IV* los resultados del caso de dos procesos que comparten un recurso y se turnan para su uso. Esto requiere que los procesos se sincronicen de forma tal que cada uno espere a que el otro libere el recurso antes de ejecutar la sección de código crítica (donde es necesario utilizar el recurso compartido, ya sean datos, hardware, etc.). Este mecanismo de control de acceso, llamado sincronización, se usa tanto en sistemas monocore como en sistemas multicore y distribuidos.

TABLA IV
MEDIDAS DE UN SEMÁFORO API VS PETRI

	Linux API	**Procesador de Petri**
Sincronización Mono-Núcleo	2.278	15.509
Sincronización en Multi-Núcleo	3.878	10.418
Semáforos Mono-Núcleo	1.817	10.681
Semáforos Multi-Núcleo	3.742	6.191

También se observa el caso de dos procesos compitiendo por un recurso que sólo puede accederse por un proceso a la vez, este es el típico caso de exclusión mutua. Se controla el acceso utilizando un semáforo.

B. Problema de la Cena de los Filósofos

La cena de los filósofos [12] es un problema clásico de las ciencias de la computación, propuesto por Edsger Dijkstra en 1965 para representar el problema de la sincronización de procesos en un SO.

Primero se lo implementó en lenguaje C++, utilizando monitores con semáforos y un proceso por cada filósofo. Esta prueba se realizó variando la afinidad de los procesos con los procesadores. Luego de ejecutar mil cenas en un monocore, se obtuvo una media de 6.524,29 ciclos de reloj por filósofo con una desviación de 532 ciclos. Para el caso de multicore, se obtuvo una media de 39.328,34 ciclos de reloj con una desviación de 5.647,40 ciclos.

Luego se implementó el problema utilizando Redes de Petri y se ejecutó en el PP usando el driver diseñada. Luego de ejecutar mil cenas en un monocore, se obtuvo una media de 50.778,06 ciclos de reloj por filósofo con una desviación de 18.878,79 ciclos. Para el caso multicore, se obtuvo una media de 18.539,59 ciclos de reloj con una desviación de 185,00 ciclos.

TABLA V
RESULTADOS DEL PROBLEMA DE LOS FILÓSOFOS.

Arquitectura	**SO**	**Petri**
Monocore	6.524	50.778
Multicore	39.328	18.539

El PP nos permitió implementar el problema de los filósofos de una manera sencilla, sin necesidad de utilizar semáforos. Esto es una ventaja ya que permitio disminuir los tiempos de diseño, desarrollo y validación. Además, los resultados observados para el caso multicore (*Tabla V*) muestran una mejora en la velocidad de resolución de concurrencia, superior al doble. Esto nos permite afirmar que nuestro procesador es adecuado para problemas que reuieren una relación sincronización/computo alta en arquitecturas multicore.

C. Extrapolar los resultados con PP en L2

Si las lecturas y escrituras de los registros del PP estuvieran a la altura de la L2, los tiempos de lectura y escritura serían tantas veces menor como la relación que existe entre la latencia de la FPGA y la OCM.

Por otro lado, notamos que el PP, posee una instrucción para quitar el disparo ejecutado sobre una transición, lo cual no es necesario cuando se sale de la sección crítica, lo cual ha sido implementado y medido. Por lo que el impacto que tiene esta mejora en el PP se muestra en la figura..

Por último, graficamos ambos escenarios juntos.

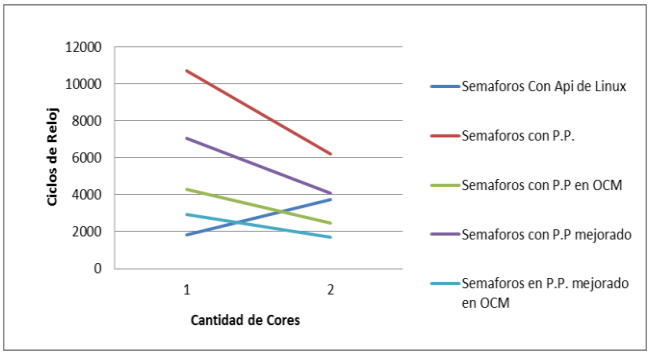

Fig. 4. Resultados obtenidos de extrapolar los resultados

VIII. CONCLUSIONES

Se logró abstraer todas las funcionalidades del procesador de Petri, facilitando tanto su programación, como su uso. Las abstracciones del diseño del driver, ha permitido usarlo con tres IP Cores distintos y siempre de caracteres. Esto ha permitido extender su uso a otros IP Cores, orientados a caracteres y con requerimientos de concurrencia.

Las mediciones de tiempo en ambientes multicores y heterogéneos son de gran complejidad. Es por ello que en este trabajo fue necesario desarrollar mecanismos específicos para la medición de performance.

En cuanto a desempeño, se observó para todos los casos evaluados que el uso del PP no ofrece ventajas con respecto a los mecanismos clásicos de sincronización, puesto que el acceso a la FPGA requiere mayor cantidad de ciclos de reloj que el acceso a la memoria. No obstante, se observó que la cantidad de ciclos de reloj necesarios para resolver problemas de concurrencia utilizando los mecanismos de sincronización de la API de GNU/Linux en arquitecturas multicore aumenta exponencialmente, mientras que para el PP es lineal.

Luego de esta investigación se introdujeron mejoras en el PP, siendo la más importante la introducción de un registro de etiquetas que evita solicitudes de disparo innecesarias. Esto nos permitió obtener mejoras del 33% en el desempeño, igualando en performance a los mecanismos de sincronización tradicionales.

Se realizaron simulaciones que determinaron que el lugar óptimo para colocar el PP es la cache L2. Los cálculos de performance obtenidos con esta latencia, resultan en un mejor desempeño al compararlo con los sistemas de sincronización tradicionales.

Asimismo, este desarrollo permite introducir un IP Core con funcionalidades específicas sin la necesidad de realizar cambios en las arquitecturas de los procesadores. En nuestro caso, nos permitió realizar la programación directa de la sección secuencial del problema, resolviendo la sección paralela y concurrente, mediante la transferencia de la matriz de adyacencia al PP.

REFERENCIAS

[1] N. G. M. Pereyra, M. Alasia and O. Micolini, "Heterogeneous Multi-Core System, synchronized by a Petri Processor on FPGA," *IEEE LATIN AMERICA TRANSACTIONS*, vol. 11, pp. 218-223, 2013.

[2] A. R. Jessica McKellar, Jonathan Corbet , Greg Kroah-Hartman, *Linux Device Drivers*, 2014.

[3] Xilinx, "Zynq-7000 All Programmable SoC," 2014.

[4] S. S. B. Sundararajan Sriram, *EMBEDDED MULTIPROCESSORS, Scheduling and Synchronization*. Boca Raton, 2009.

[5] R. A. B. David R. Martinez, M. Michael Vai, *High Performance Embedded Computing Handbook A Systems Perspective*. Massachusetts Institute of Technology, Lincoln Laboratory, Lexington, Massachusetts, U.S.A.: CRC Press, 2008.

[6] ARM, "AMBA Design Kit Technical Reference Manual," 2007.

[7] ALTERA, "PCI Express High Performance Reference Design," 2014.

[8] M. Domeika, *Software Development for Embedded Multi-core Systems*. 30 Corporate Drive, Suite 400, Burlington, MA 01803, USA, Linacre House, Jordan Hill, Oxford OX2 8DP, UK, 2008.

[9] A. J. A. W. y. K. QIAN, *Component Oriented Programming*. New Jersey, 2005.

[10] J. N. Micolini Orlando, Carlos R. Pisetta, "IP Core for Timed Petri Nets," *CASECongreso Argentino de Sistemas Embebidos*, pp. 3-8, 2013.

[11] J. N. y. C. R. P. Ornaldo Micolini, "IP Core Para Redes de Petri con Tiempo," *CASIC 2013*, pp. 1097-110, 2013.

[12] K. M. M. Chandy, J., "The Drinking Philosophers Problem," *ACM Transactions on Programming Languages and Systems*, 1984.