MatrixmodG: Driver para gestionar Redes de Petri Generalizadas en el kernel de Linux

Dr. Orlando Micolini, Ing. Gabriel Sosa Ludueña

Laboratorio de Arquitectura de Computadoras, FCEFyN, UNC, Córdoba, Argentina

omicolini@compuar.com
gsosaludu@gmail.com

Abstract. Este artículo presenta un enfoque diferente para la gestión de Redes de Petri Generalizadas (RDPG) desde el espacio usuario, proponiendo el uso de múltiples hilos que interactúan con un Device Driver de Linux, denominado MatrixmodG, desde donde se realiza la gestion de la RDPG. El driver se ejecuta en el kernel con la característica de multiprocesamiento simétrico seguro (código SMPs), permite gestionar RDPG a cualquier aplicación de espacio usuario, independientemente del lenguaje de programación a utilizar, y provee mejoras de rendimiento al espacio usuario trabajando en conjunto con el kernel para la gestión RDPG. El driver se comporta como una nueva funcionalidad del sistema operativo Linux, brindando disponibilidad a cualquier aplicación del mismo. Para cumplir con el enfoque propuesto se desarrolló una librería de C++ que permite la comunicación de subprocesos con el driver en las gestiones de la RDPG cargada en el kernel, a través de llamadas al sistema, llevando así todo su rendimiento al espacio usuario. De esta forma, se obtuvo una herramienta que gestiona subprocesos con ayuda del kernel, y simula o resuelve problemas de concurrencia complejos modelados con RDPG. Se aborda un caso de aplicación, que hace uso de múltiples hilos, y demuestra las ventajas de gestionar RDPG con el driver a comparación de una aplicación de usuario de características similares.

Keywords: Red de Petri Generalizada, Device Driver Linux, Ambiente Multicore, Múltiples hilos, Programación Concurrente.

1 Introducción

En su tesis doctoral "Comunicación con autómatas", Karl Adam Petri en 1962, estableció las bases para una teoría de comunicación entre componentes asíncronos de un sistema de cómputo. Definió una herramienta matemática de propósito general para describir las relaciones que existen entre condiciones y eventos, el resultado fueron las Redes de Petri no autónomas [1].

Tanto por el alcance de los resultados teóricos como por la diversidad y el número de aplicaciones, las Redes de Petri (RdP) constituyen en la actualidad un modelo formal avanzado y completo para la descripción lógica de las estructuras de control del paralelismo y la concurrencia de un sistema modelado [2].

Las RdP conforman una herramienta gráfica y matemática que puede aplicarse especialmente a los sistemas paralelos que requieran simulación y modelado de la

concurrencia en los recursos compartidos, de la misma manera que lo hace los autómatas finitos en los sistemas secuenciales [3].
Sin embargo, la implementación de hilos en los sistemas informáticos permite explotar los recursos de las arquitecturas multicore [4]. Estos hilos cooperan y se ejecutan concurrentemente. Las aplicaciones diseñadas de esta manera tienen una complejidad intrínseca adicional con respecto a los programas secuenciales. Esta complejidad se manifiesta en el diseño, detección de errores, testing, validación y mantenimiento [5]. Por lo mencionado, es necesario introducir mecanismos de control que no penalicen los tiempos de ejecución y provea una solución formal que garantice y facilite el desarrollo e implementación del sistema.
Para la simulación y ejecución de sistemas complejos, investigaciones como [6] proponen el uso de un modelo de RdP extendido, el cual, amplía la semántica y capacidad de expresión de las RdP con la inclusión de eventos, guardas, distintos tipos de brazos (inhibidores, lectores y reset) y semánticas temporales [7] optimizando el modelo para tener mayor control del dominio del problema, de mayor tamaño y más complejos. De esta forma se propone el uso de las Redes de Petri Generalizadas (RDPG) para la solución y simulación de sistemas más complejos, brindando un modelo matemático potente y adaptable a todos los sistemas de computación.
En el presente artículo se analizan los puntos claves del desarrollo de un Device Driver de Linux (DDL) para gestionar RDPG en el kernel, en conjunto con su librería de espacio usuario de C++. El driver abarca las extensiones de las RDPG relacionadas a los arcos inhibidores, lectores y reset en conjunto con la extensión de guardas sobre las transiciones. Con la librería C++ se permite simular la extensión de una semántica temporal y la extensión referida a los eventos se plantea como una mejora del proyecto. Este nuevo enfoque brindara una herramienta de programación que implementa toda la lógica del sistema en el kernel para gestionar RDPG en programas de espacio usuario que requieran mayor rendimiento facilitándose la disposición del estado global del sistema modelado a todo el espacio usuario para una o múltiples aplicaciones.

2 Redes de Petri

2.1 Estructura matemática de una RdP

Una RdP se define matemáticamente como una 4-tupla definida por RdP = (P, T, I, M_0). Donde el significado de cada elemento es el siguiente:

- P = {p_1, p_2, ..., p_n} es un conjunto finito y no vacío. Representa las plazas y su cardinalidad es n.
- T = {n_1, n_2, ..., n_n} es un conjunto finito y no vacío. Representa las transiciones y su cardinalidad es m.
- I = es una función de incidencia de todas las salidas y entradas por cada plaza y transición de la red. Es la función de las relaciones dadas por los arcos entre las plazas y transiciones y viceversa. La función representa el mapeo de todos los arcos entre plazas y transiciones. Esta función se representa por una matriz denominada matriz de incidencia I que relaciona las n plazas con las m

transiciones en una matriz de dimensión nxm, en donde cada elemento a_{ij} toman los siguientes valores:

- $a_{ij} = 0$, si entre pi hacia t_j o t_j hacia pi no existe relación de arco.
- $a_{ij} = W_{ij}$, si entre t_j hacia p_i existe relación de arco.
- $a_{ij} = -W_{ij}$, si entre p_i hacia t_j existe relación de arco.

Donde W_{ij} es el peso del arco que une la plaza pi con transiciones t_j o viceversa.

- M_0 es el marcado inicial de la red, se representa como un vector de asignación de recursos a las plazas de la red, de esta forma se define la configuración inicial de los tokens de la red. Por ejemplo puede definirse el marcado de una plaza como M[i], esto indicaría la cantidad de tokens ubicados en la plaza "i".

2.2 Dinámica de una RdP

Definida la estructura matemática de una RdP, ahora queremos introducir el comportamiento dinámico de esta, el cual está determinado por la habilitación y disparo de sus transiciones, lo que se explica a continuación.

Las reglas de ejecución de una RdP son:

- Una RdP se ejecuta por los disparos de sus transiciones.
- Para que una transición pueda ser disparada debe estar habilitada (o sensibilizada).

Una transición está sensibilizada si cada una de las plazas de entrada a la transición tiene al menos la cantidad de token indicados en el peso, que está en el arco que une la plaza con la transición.

Disparar cualquiera de las transiciones de una RdP implica avanzar a un nuevo estado o permanecer en el estado actual, de acuerdo a si la transición a disparar esta sensibilizada o no lo está respectivamente. La ecuación que define la dinámica de una RdP cuando se dispara una transición se denomina ecuación de estado de la RdP y se define de la siguiente forma.

$$\boldsymbol{M_{i+1} = M_i + (I\ x\ \sigma_i)}$$

Donde Los componentes son:

- M_{i+1}: Nuevo estado a alcanzar de acuerdo al disparo dado por el vector U_i.
- M_i: Estado actual de la RdP en el instante i.
- I: Matriz de incidencia I.
- σ_i: Vector disparo responde a la función uno, donde todos sus componentes son nulos a excepto la del disparo a realizar en el instante i.

Esta ecuación de estado responde de manera ideal si se dispara una transición sensibilizada o no sensibilizada, brindando información en ambos casos para determinar un nuevo estado de la RdP si se disparó una transición sensibilizada o permanecer en el mismo estado si se disparó con una transición no sensibilizada.

Es posible que en una RdP se tengan distintas transiciones listas en el mismo instante de tiempo. Las transiciones listas en este caso son concurrente, el sistema de control de disparo es quien realiza la decisión de cual transición se dispara [8].

2.3 Transición sensibilizada

Una transición está sensibilizada si todos las plazas de entrada a la transición tienen una marca igual o mayor al peso del arco que une cada plaza con la transición. Esto se expresa como:

$$\boldsymbol{M(p_i) \geq W_{ji}, \forall\, p_j \in I(t_i)}$$

Las transiciones sensibilizadas se expresan como un vector binario E de dimensión $m \times 1$. Cada componente del vector indica con uno la transición sensibilizada y un cero la que no.

$$\boldsymbol{E = \left(sign(S^0), sign(S^1), \dots, sign(S^{m-1}) \right)}$$

Donde la relación ( ) de cada vector Si es:

- S^i es igual a $Mj + I^i$, un vector con el estado que se obtendría de disparar la i-ésima transición una vez.
- I^0, I^1,... , I^{m-1}, son las columnas de la matriz I.
- sign(S^i) , es binario siendo cero si alguna de las componentes de S^i es negativa, de otra forma es uno.

3 Redes de Petri Generalizadas

3.1 Ecuación de estado extendida

Teniendo en cuenta todas las ampliaciones de una RdP a la semántica expuesta por [6], se destaca que siempre se trata de determinar cuáles transiciones están sensibilizadas. Una vez determinada la transición a disparar, el disparo se realiza con la ecuación de estado original, excepto para el brazo reset que retira todos los token de la plaza. Para representar matemáticamente la existencia de los brazos enumerados anteriormente se requiere de una matriz para indicar la conexión plaza transición.

Estas matrices son similares a la matriz I. Los términos de las matrices son binarios, dado que los pesos de los arcos son uno. Es decir, para que una transición esté sensibilizada se requiere:

- Si tiene brazo inhibidor que la plaza no tenga token.
- Si tiene brazo lector que la plaza tenga uno o más token
- Si tiene guarda que el valor de la guarda sea verdadero.
- Si tiene evento que tenga uno o más eventos en la cola asociada.
- Si tiene etiqueta con intervalo de tiempo, que el contador se encuentre en el intervalo.

De esta forma la nueva ecuación de estado considera un nuevo conjunto de matrices y vectores para el nuevo modelo matemático. Tal como propone [6] para disparar una transición se requiere la conjunción lógica entre todas las condiciones enumeradas anteriormente y el vector E de transiciones sensibilizadas.

- El vector de transiciones des-sensibilizadas por arco inhibidor $\boldsymbol{B}$.
- El vector de transiciones des-sensibilizadas por arco lector $\boldsymbol{L}$.

- El vector de transiciones des-sensibilizadas por guarda G.
- Vector de transiciones des-sensibilizadas por evento $\boldsymbol{V}$.
- El vector de transiciones des-sensibilizadas por tiempo $\boldsymbol{Z}$.
- El vector de transiciones reset $\boldsymbol{A}$.

Con todos los vectores enumerados, se determina el vector de sensibilizado extendido $\boldsymbol{Ex}$ de una RDPG, que se obtiene de la conjunción lógica de todos los vectores.

$$\boldsymbol{Ex = E\ and\ B\ and\ L\ and\ V\ and\ G\ and\ Z}$$

Para introducir el brazo reset hay que multiplicar elemento a elemento (#) a l vector que resulte de la conjunción por A. Por lo que la ecuación de estado extendida resulta:

$$\boldsymbol{M_{i+1} = M_i + I\ x\ ((\sigma\ and\ Ex)\ \#\ A)}$$

La importancia de esta ecuación de estado es que es de simple implementación como circuito combinacional en una FPGA.

4 Características de driver MatrixmodG

El proyecto MatrixmodG se encuentra disponible en un repositorio remoto, su enlace es el siguiente: https://github.com/gslAgile/Proyecto-MatrixmodG.

4.1 Diseño POO en C

Los DDL son programados en lenguaje C a bajo nivel, por lo que no se cuenta con demasiado soporte como lo es en el espacio usuario. Sin embargo Linux sigue evolucionando para que su programación sea más accesible. Uno de sus puntos fuertes son la compilación mediante los makefile que trae numerosas funcionalidades potentes, y existen variedades de herramientas y frameworks para el kernel que se complementan para obtener mejor calidad en el diseño de los DDL como son el framework KTF (Kernel Test Framework) y KEDR (KErnel-mode Drivers in Runtime).

Además, como mejora de diseño para la creación del DDL MatrixmodG se decidió utilizar un conjunto de patrones de diseño del kernel de Linux que afectan a un estilo de programación orientado a objetos (POO) tal como propone [9]. Esta mejora impacto positivamente para la creación del driver, como también para los procesos de prueba y mantenimiento dando como resultado un producto de mayor calidad con atributos claves como portabilidad, mantenibilidad, escalabilidad y eficiencia.

4.2 Arquitectura de alto nivel: Vista lógica

El diagrama de clase del DDL MatrixmodG en su forma más simple es el de la figura 4.1 donde muestra la relación de las diferentes clases del diseño realizado, sus asociaciones y las diferentes enumeraciones asociadas a cada clase.

La clase principal es RDPG_o, esta clase es la que representa a un objeto RDPG al momento de instanciarse. En el diagrama de la figura 4.1 se puede observar las asociaciones de composición (arcos con extremo de rombo negro) que se desprenden de la clase RDPG_o hacia tres tipos de clases, que son la clase matrix_o, vector_o y

rwlock. Estas últimas clases representan los objetos matrices, vectores y spinlock lector-escritor por los que se compone una RDPG_o al momento de instanciarse.

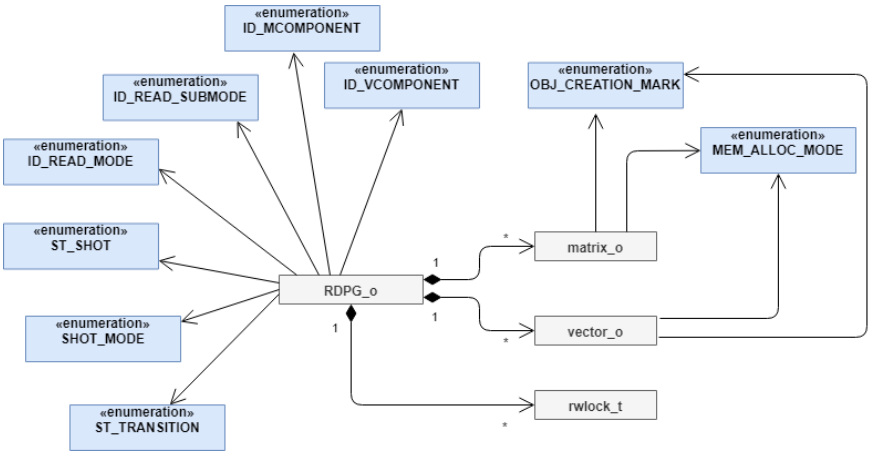
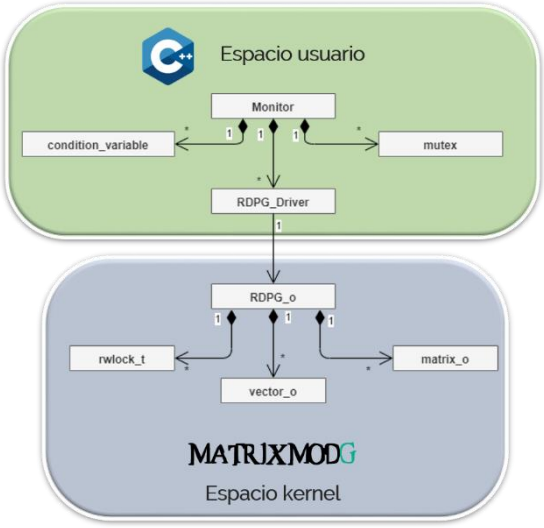


Figura 4.1 Diagrama de clase de manera general.

Figura 4.2 Diagrama de clases desde espacio usuario.



La figura 4.2 muestra una nueva abstracción de la arquitectura con el uso de monitores y sus componentes desde el espacio usuario.

La idea del diagrama de clase de la figura 4.2 es mostrar cómo se tratan las RDPG desde el espacio usuario. La clase principal en el espacio usuario es el Monitor y sus componentes. Los componentes del monitor son el mutex (encargado de proveer exclusión mutua a los procesos) y las variables de condición (quienes proveen las diferentes colas de procesos necesarias para una RDPG). Existe un único mutex dentro del monitor, ya que en el monitor, solamente un proceso puede operar con el recurso compartido. Las variables de condición coinciden con el número de transiciones de la RDPG que se esté utilizando, es decir que se trata de un vector de variables de condición, y representan las diferentes colas en donde puede dormirse un proceso.
Además desde el espacio usuario se permite que el driver sea accedido y controlado por múltiples aplicaciones de lenguajes diferentes al mismo tiempo. Esto significa que dos o más aplicaciones de diferentes lenguajes pueden estar operando con una misma RDPG en el kernel mediante el uso del driver MatrixmodG, este nuevo paradigma de gestionar

una RDPG del kernel por múltiples aplicaciones de lenguajes diferentes se muestra en la figura 4.3.

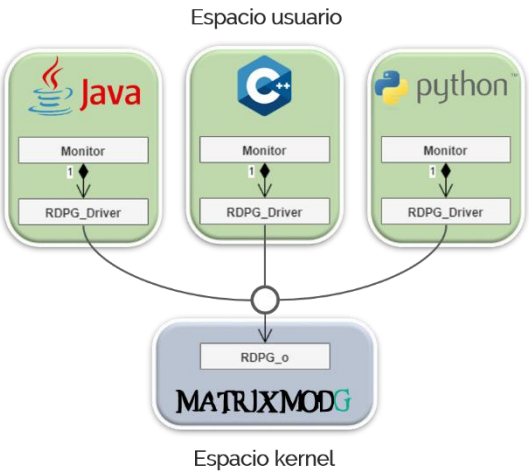


Figura 4.3 Extensión diagrama de clases a múltiples aplicaciones de lenguajes diferentes.

4.3 Asignación de memoria dinámica y soporte SMPs

Para el DDL MatrixmodG se estudió y analizo un conjunto de funciones de las familias de kmalloc y vmalloc que fueron las funciones que se buscaban para el diseño propuesto, haciendo un estudio del rendimiento de cada una de las alternativas y seleccionando la alternativa de mayor rendimiento por defecto pero permitiendo también trabajar con la selección del resto de alternativas.

El bloqueo más común en el kernel de Linux es el spin lock (bloqueo de giro). Un bloqueo de giro es un bloqueo que puede mantener como máximo un subproceso de ejecución. Si un subproceso de ejecución intenta adquirir un bloqueo de giro mientras ya está retenido, lo que se denomina contenido , el subproceso está ocupado, gira, esperando a que el bloqueo esté disponible [10]. Este mecanismo es el que se utilizó en el DDL MatrixmodG para proteger el recurso compartido (RDPG) entre los subprocesos. Como mejora se hizo uso de un tipo de spin lock especial para problemas lector-escritor que es el caso de las RDPG, siempre se leen y escriben, de esta forma se agregó mayor eficiencia en la gestión del spin lock.

5 Caso de aplicación

5.1 Procesador dual-core modelado con RDPG

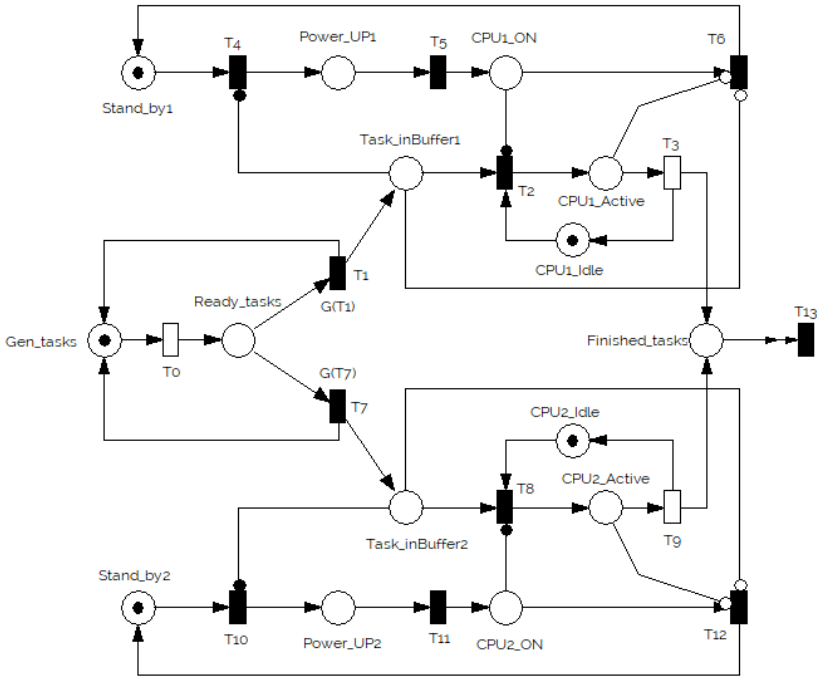


Figura 5.1 Modelo de RDPG para procesador dual-core.

El caso de aplicación de interés se basa en la simulación de una RDPG que represente el modelo de los estados de un procesador dual-core respecto al procesamiento de tareas y que impacta en su ahorro energético. Para esto se extiende la RDPG del procesador mono-core a un procesador dual-core, donde se administra y controla cada CPU del procesador en busca de un comportamiento eficiente en la gestión de tareas permitiendo su ahorro energético cuando se acaben todas las tareas, tal como lo hace el procesador mono-core propuesto por [11].

La figura 5.1 muestra el modelo de la RDPG del procesador dual-core. En el modelo se nota que cada tarea generada puede ser procesada por la CPU1 o la CPU2 del procesador, para controlar esto se debe definir una política para controlar con guardas sobre las transiciones T1 y T7 (G(T1) y G(T7)).

Tabla 5.1

Plaza	Modo de uso	Descripción
Gen_tasks	Estado de tarea	Representa las tareas generadas.
Ready_tasks	Estado de tarea	Representa las tareas en su estado listas para enviar a la CPU.
Stand_by1	Estado de CPU	Representa la CPU1 en estado de espera (Stand_by1). Es el estado que permite ahorrar energía a la CPU1.

Power_up1	Estado de CPU	Representa la CPU1 en estado previo a su encendido. Es cuando la CPU1 tiene solicitud o aviso de encendido.
CPU1_ON	Estado de CPU	Representa la CPU1 en estado encendida. En este estado la CPU1 ya puede empezar a procesar tareas.
Task_inBuffer1	Estado de tarea	Representa las tareas que se encuentran en el buffer de la CPU1. Son las tareas listas para ser procesadas por la CPU1.
CPU1_active	Estado de CPU y tarea	Representa la CPU1 en estado activa mientras la CPU1 se encuentra encendida. Es el estado que representa el procesamiento de una tarea.
CPU1_idle	Estado de CPU y tarea	Representa la CPU1 en estado desocupado mientras la CPU1 se encuentra encendida.

Tabla 5.2

Transición	Tipo de transición	Descripción
T0	Temporal	Representa la acción de pasar las tareas a su estado de tareas listas para enviar a la CPU correspondiente.
T1	Con guarda	Representa la acción de enviar las tareas al buffer de la CPU1. Si la guarda de la transición es verdadera la acción de la guarda se llevara a cabo, si es falsa no se lleva a cabo.
T2	Inmediata	Representa la acción de pasar las tareas a su procesamiento en la CPU1 y la CPU1 al estado activo.
T3	Temporal	Representa la acción de pasar la CPU1 al estado desocupado (idle). La transición es temporal, representa el tiempo de procesamiento de cada tarea.
T4	Inmediata	Representa la acción de pasar la CPU1 al estado de petición de encendido.
T5	Inmediata	Representa la acción de pasar la CPU1 al estado encendido.
T6	Inmediata	Representa la acción de pasar la CPU1 al estado de espera (Stand_by1).
T13	Inmediata	Representa la acción de limpiar el número de las tareas procesadas por la CPU1 y CPU2 del procesador.

Los detalles de las plazas y transiciones de la CPU1 se describen en las tablas 5.1 y 5.2. No se describen los detalles de las plazas y transiciones de la CPU2 ya que por analogía son iguales a los detalles de la CPU1 respectivamente. Por ultimo para explotar todas las capacidades de expresión de las RDPG que gestiona el módulo MatrixmodG, se hace uso de un arco reset para vaciar la plaza “Finished_tasks” que representa todas las tareas finalizadas por el procesador.

5.2 Gestión de la política

Las condiciones que determinan la política son las siguientes:

- Si M(Tasks_inBuffer1) <= M(Tasks_inBuffer2): Procesara la tarea la CPU1, se inhibe la transición T7. M(Tasks_inBuffer1) es la marca de la plaza Task_inBuffer1 y que representa la cantidad de tareas en el buffer de la CPU1. Análogamente M(Tasks_inBuffer2) es la marca de la plaza Task_inBuffer2.

- Si M(Tasks_inBuffer1) > M(Tasks_inBuffer2): Procesara la tarea la CPU2, se inhibe la transición T1.

Estas dos condiciones controlan las guardas que se utilizan sobre las transiciones que envían las tareas a cada CPU (G(T1) y G(T7)), de acuerdo con la figura 5.1 son la transición T1 y la transición T7. De esta forma la RDPG permite controlar la política para la gestión de tareas y los propios CPUs.

6 Resultados

6.1 Comparación de DDL MatrixmodG vs APP C++

Realizando la comparación de los tiempos de las operaciones más importantes para la gestión de RDPG y comparando los tiempos obtenidos por el driver MatrixmodG contra los tiempos obtenidos por la APP C++, se determina la mejora porcentual de rendimiento que produce el uso del driver MatrixmodG.

Dentro de todas las operaciones se analiza la operación de disparo de transiciones ya que es una de las más importantes para la gestión de las RDPG.

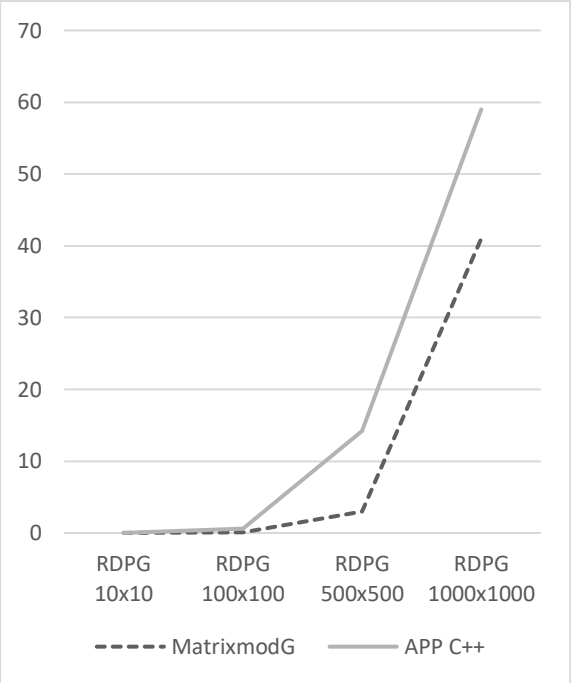
La comparación se realiza en la tabla 6.1. Se utilizan los tiempos medidos por la API OpenMP.

La gráfica 6.1 muestra la mejora de la operación de disparo de transiciones, solapando las dos curvas de tiempo para cada uno de los sistemas comparados.

Tabla 6.1: Comparación Op disparo de transición

Tiempo de Operación	Driver MatrixmodG	APP C++	Mejora Porcentual
RDPG 10x10	0,01 ms	0,01 ms	0%
RDPG 100x100	0,1 ms	0,6 ms	80%
RDPG 500x500	3 ms	14,2 ms	75%
RDPG 1000x1000	41 ms	59 ms	30%
Mejora Porcentual promedio			**45%**

Grafica 6.1 Comparación Driver-APP operación de disparo de transición.



Analizando el rendimiento de la gestión de la RDPG del caso de aplicación asociado al procesador dual-core, gestionando la RDPG desde el driver MatrixmodG y desde la APP C++. Como el caso de aplicación gestiona tareas a procesar por los diferentes cores del procesador, se expuso la RDPG de acuerdo a un modo de ejecución con 5 hilos concurrentes para un conjunto de tareas que inician desde las 10 hasta las 100.000 tareas, los resultados se registraron en la tabla 6.2.

Tabla 6.2: Comparativa de tiempos de gestión de RDPG de Caso de Aplicación con 5 hilos concurrentes.

Tareas	MatrixmodG 5 hilos	APP C++ 5 hilos	Mejora porcentual
10	0,9 ms	1,1 ms	20%
100	5 ms	8 ms	37%
1000	47 ms	74 ms	36%
10000	455 ms	704 ms	35%
100000	4600 ms	7100 ms	35%
Mejora porcentual promedio			**33%**

La tabla 6.2 muestra que gestionar RDPG en el driver matrixmodG tiene mayor rendimiento que realizarlo directamente desde el espacio usuario y mayor es la mejora a medida que más trabajo se realiza sobre la RDPG.

7 Conclusiones

En este trabajo quedo demostrado que el DDL MatrixmodG sirve como una herramienta totalmente independiente del lenguaje de programación, ya que es una extensión del kernel y se la puede entender como una herramienta del sistema operativo en general, la cual permite que múltiples aplicaciones de usuario resuelvan en conjunto un mismo problema de concurrencia, garantizando el control completo de todos los subprocesos involucrados. La creación de una librería de espacio usuario asociada, posibilita que la capa de aplicación obtenga todo el rendimiento del driver dentro de la programación de un lenguaje particular, demostrándose sobre el lenguaje C++. Además el driver puede

utilizarse con el paradigma de gestionar la RDPG por múltiples aplicaciones de diferentes lenguajes, donde se debe prestar atención a que no está resuelta la notificación del estado global del sistema simulado para todas las aplicaciones que operan con el driver, es decir que se presenta el problema de que solo los subprocesos, del ultimo lenguaje que peticiona al driver, serán los procesos notificados al cambio del estado global del sistema modelado, ignorándose la notificación de las aplicaciones, y subprocesos asociados, del resto de lenguajes involucrados a la misma RDPG. Para esto se propone la generación de una nueva versión del driver que disponga de un conjunto de archivos de dispositivo extras asociados a cada aplicación para realizar la notificación a "subprocesos controladores" encargados de actualizar el estado global del sistema, en cada aplicación de un lenguaje de manera particular, cuando exista un cambio de estado en el sistema simulado mediante la RDPG del kernel.

Los resultados de rendimiento obtenidos sobre el DDL MatrixmodG para gestionar RDPG en el kernel de Linux, demuestran mejoras significativas en comparación a los resultados obtenidos por la aplicación de espacio usuario (aplicación C++). Desde varios casos de aplicación ejecutados se pudo verificar que para el caso una RDP pequeña (menos de 10 plazas y transiciones) no se detectan mejoras de rendimiento. Sin embargo para el caso de aplicación analizado, de una RDPG mediana (más de 10 plazas y transiciones), se detectan mejoras significativas del 30% en promedio para la operación de disparos de transiciones.

De esta manera se puede concluir que la gestión de Redes de Petri Generalizadas desde el driver MatrixmodG tiene un mejor rendimiento que su gestión directa desde una aplicación de usuario y mayor será la mejora mientras más grande sea la red utilizada.

La creación de la librería C++ asociada al driver MatrixmodG, para la gestión de RDPG en el kernel de Linux desde el espacio usuario, demuestra ser una librería de fácil uso, simple, entendible y adaptable a las necesidades que puede requerir un usuario final, realizando todas las operaciones con el driver MatrixmodG de una forma eficiente y totalmente transparente, gracias a la gestión de las RDPG como un objeto de espacio usuario. De esta forma la librería del driver MatrixmodG es una nueva herramienta que permite la simulación y/o control de problemas de concurrencia complejos mediante la gestión de RDPG desde programas C++ con protección de recursos compartidos sometidos al multiprocesamiento, para el uso de múltiples hilos, y brindando un alto rendimiento.

Referencias

1. David R, Alla H. Petri nets and Grafcet: tools for modelling discrete event systems. 1992.
2. Brams GW. Las redes de Petri: teoría y práctica. Modelización y aplicaciones: Masson; 1986.
3. Desel J, Esparza J. Free choice Petri nets: Cambridge university press; 2005.
4. Martinez DR, Bond RA, Vai MM. High performance embedded computing handbook: A systems perspective: CRC press; 2008.
5. Domeika M. Software Development for Embedded Multi-core Systems: A Practical Guide Using Embedded Intel Architecture: Elsevier Science; 2011.

6. Micolini Orlando, Cebollada M, Eschoyez M, Ventre Luis O, Schild M, editors. Ecuación de estado generalizada para redes de Petri no autónomas y con distintos tipos de arcos. XXII Congreso Argentino de Ciencias de la Computación (CACIC 2016); 2016.
7. Diaz M. Petri nets: fundamental models, verification and applications: John Wiley & Sons; 2013.
8. Iordache M, Antsaklis PJ. Supervisory control of concurrent systems: a Petri net structural approach: Springer Science & Business Media; 2007.
9. Neil Brown. Object-oriented design patterns in the kernel. (June 1, 2011). Accedido a la fecha 28/01/2019. Consultado desde: https://lwn.net/Articles/444910/.
10. Love Robert. Linux Kernel Development: Linux Kernel Development: Pearson Education; 2010.
11. Shareef Ali, Zhu Yifeng, editors. Energy modeling of wireless sensor nodes based on Petri nets. 2010 39th International Conference on Parallel Processing; 2010: IEEE.