UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

Escola Tècnica
Superior d'Enginyeria
Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Mejorar las prestaciones del prefetcher utilizando técnicas de deep learning

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Autor: Manel Lurbe Sempere

Tutores: Julio Sahuquillo Borrás
Salvador Vicente Petit Martí

Director experimental: Josué Feliu Pérez

Curso 2018-2019

Glosario

API Constituye una interfaz de programación de aplicaciones (*Application Programming Interface*). Es un conjunto de rutinas que provee acceso a funciones de un determinado software. 27, 35, 50

array Vector de elementos.. 40

bash Programa informático, cuya función consiste en interpretar órdenes, y un lenguaje de consola. 50

benchmark Consiste en una prueba de rendimiento o comparativa. Es una técnica utilizada para medir el rendimiento de un sistema o uno de sus componentes, y poder comparar los resultados con máquinas similares. 25, 27, 29, 32, 41, 51

caché Componente de hardware que almacena datos para que las solicitudes futuras de esos datos se puedan atender con mayor rapidez. 1, 2, 11–13, 19, 24

deep learning Conjunto de algoritmos de aprendizaje automático que intenta modelar abstracciones de alto nivel en datos usando arquitecturas computacionales que admiten transformaciones no lineales múltiples e iterativas de datos expresados en forma matricial o tensorial. 5, 11, 14–16, 50

DSCR Data Streams Control Register. Registro de control del prefetch en procesadores IBM POWER8. 20–22, 27, 29, 34, 35, 39, 40

eDRAM *embedded DRAM* (o «DRAM embebida»).Tipo de memoria dinámica de acceso aleatorio basada en condensadores que se integra en el mismo encapsulado que un microprocesador. 12, 20

framework Entorno de trabajo o marco de trabajo, que consta de un conjunto estandarizado de conceptos, prácticas y criterios para enfocar un tipo de problemática particular que sirve como referencia, para enfrentar y resolver nuevos problemas de índole similar. 8

funciones de loss Mide que tan insatisfechas resultan las predicciones de un modelo de *machine learning* con respecto a una respuesta correcta. 17, 36, 41

hardware Conjunto de elementos físicos o materiales que constituyen una computadora o un sistema informático. 2–4, 7, 8, 13, 19, 24, 25, 30, 35, 49, 50

hit Acierto. En el ámbito de memorias de computadores,cuándo una aplicación solicita un dato y este se encuentra en una memoria caché. 11, 13

kernel de Linux Corresponde al núcleo del sistema que se encarga de manejar los recursos hardware, en este caso se llama Linux. 19

Ley de Moore Expresa que aproximadamente cada dos años se duplica el número de transistores en un microprocesador. 1

LLC *Last level cache*. Corresponde a la denominación de la caché de último nivel en las jerarquías de memoria. 2, 12

machine learning Aprendizaje automático. Estudio científico de algoritmos y modelos estadísticos que los sistemas informáticos utilizan para realizar una tarea específica sin utilizar instrucciones explícitas, sino que se basan en patrones e inferencias.. 1–5, 13–16, 27, 35, 49

miss Fallo. En el ámbito de memorias de computadores, cuándo una aplicación solicita un dato y este no se encuentra en una memoria caché. Esto genera un acceso a memoria principal o un nivel inferior de la jerarquía. 11, 13

on-chip Referido a los componentes que están incluidos en el propio chip. 19

overfitting Sobreajuste. En aprendizaje automático, efecto de sobreentrenar un algoritmo de aprendizaje con unos ciertos datos para los que se conoce el resultado deseado. 17, 38

PID *Process Identifier*. Identificador de proceso dentro de un sistema informático. 22

prefetch Prebúsqueda. 3–5, 7, 13, 20, 22, 25, 26, 29–32, 40, 42–47, 50

prefetcher Mecanismo de prebúsqueda. 2–5, 7, 8, 13, 20, 22, 24, 27, 29, 30, 46, 49

red neuronal artificial Modelo computacional vagamente inspirado en el comportamiento observado en el funcionamiento de las neuronas humanas. Consiste en un conjunto de unidades, llamadas neuronas artificiales, conectadas entre sí para transmitirse señales. La información de entrada atraviesa la red neuronal (donde se somete a diversas operaciones) produciendo unos valores de salida. 4, 5, 13–15, 27, 30–32, 35, 38

registro Memoria de alta velocidad y poca capacidad que integran los procesadores, algunos registros tienen funciones de hardware específicas y pueden ser de solo lectura o solo de escritura. 20

script Archivo de texto plano que almacena distintas ordenes de consola para facilitar la ejecución de comandos. 26, 49

shell Intérprete de órdenes o intérprete de comandos. Es el programa informático que provee una interfaz de usuario para acceder a los servicios del sistema operativo. 26

SMT Simultaneous Multithreading. Se refiere a la posibilidad de ejecutar múltiples subprocesos y tareas simultáneamente en un mismo núcleo de un procesador. 8, 19

software Representa un conjunto de programas y rutinas que permiten a la computadora realizar determinadas tareas. 8, 13, 27

SRAM *Static Random Access Memory*. Tipo de tecnología de memoria RAM basada en semiconductores, capaz de mantener los datos, mientras siga alimentada, sin necesidad de circuito de refresco. 19

Resum

Recentment hi ha hagut un gran increment d'aplicacions de les xarxes neuronals gràcies a noves tècniques d'aprenentatge profund (*deep learning*), un nou tipus d'aprenentatge automàtic que ha estat possible gràcies a l'augment de la capacitat de còmput i de l'ús de xarxes neuronals multicapa, les quals actuen com aproximadors universals que permeten modelar relacions no lineals entre dades d'entrada i sortida. En aquest projecte es pretén utilitzar les xarxes neuronals i l'aprenentatge profund per predir quina configuració de la prebúsqueda maximitza les prestacions del sistema executant aplicacions. D'aquesta manera, serà possible establir un model basat en xarxes neuronals que prediga la configuració del *prefetcher* més adequada per a cada aplicació.

Paraules clau: Deep learning, Reds neuronals, Prebúsqueda hardware, Prestacions del sistema

Resumen

Recientemente ha habido un gran incremento de aplicaciones de las redes neuronales gracias a nuevas técnicas de aprendizaje profundo (*deep learning*), un nuevo tipo de aprendizaje automático que ha sido posible gracias al incremento de la capacidad de cómputo y del uso de redes neuronales multicapa, las cuales actúan como aproximadores universales que permiten modelar relaciones no lineales entre datos de entrada y salida. En este proyecto se pretenden aprovechar las redes neuronales y el aprendizaje profundo para predecir qué configuración de la prebúsqueda maximiza las prestaciones de un sistema ejecutando aplicaciones. De esta manera, será posible establecer un modelo basado en redes neuronales que prediga la configuración del *prefetcher* más adecuada para cada aplicación.

Palabras clave: Deep learning, Redes neuronales, Prebúsqueda hardware, Prestaciones del sistema

Abstract

Recently there has been a large increase in applications of neural networks thanks to new deep learning techniques, a new type of machine learning that has been possible thanks to the increase in computing capacity and the use of multilayer neural networks, which act as universal approximators that allow modeling nonlinear relationships between input and output data. This project aims to use neural networks and deep learning to predict which pre-search configuration maximizes system performance by running applications. In this way, it will be possible to establish a model based on neural networks that predicts the most appropriate prefetcher configuration for each application.

Key words: Deep learning, Neural networks, Prefetch, System perfomance

Índice general

Índice general VII
Índice de figuras IX
Índice de tablas IX

1 Introducción 1
1.1 Descripción del problema . . . 1
1.2 Motivación . . . 2
1.3 Objetivos . . . 3
1.4 Impacto esperado . . . 3
1.5 Metodología . . . 3
1.6 Estructura de la memoria . . . 4
1.7 Convenciones . . . 5

2 Estado del arte 7
2.1 Trabajos relacionados con el *prefetch* y la jerarquía de memoria . . . 7
2.2 Trabajos relacionados con la configuración de hardware empleando *machine learning* . . . 8

3 Conceptos básicos 11
3.1 Mecanismos de reducción de la latencia de acceso a memoria . . . 11
3.1.1 Jerarquías de memoria . . . 11
3.1.2 Prebúsqueda . . . 12
3.2 Machine learning . . . 13
3.2.1 Definiciones previas . . . 13
3.2.2 Terminología básica . . . 15

4 Entorno de trabajo 19
4.1 Arquitectura del sistema . . . 19
4.1.1 Procesador IBM POWER8 . . . 19
4.1.2 Mecanismos de prebúsqueda en el IBM POWER8 . . . 20
4.1.3 Configuración del mecanismo de prebúsqueda . . . 22
4.2 Contadores de prestaciones . . . 24
4.3 *Benchmarks* SPEC CPU 2006 . . . 25
4.4 Herramientas . . . 26

5 Configuración dinámica de prebúsqueda basada en machine learning 29
5.1 Efectos de las configuraciones del *prefetcher* en cada aplicación . . . 29
5.2 Diseño de la propuesta . . . 30
5.2.1 Obtención de los datos . . . 30
5.2.2 Diseño de la red neuronal artificial . . . 35
5.3 Evaluación de la red neuronal artificial . . . 36
5.4 Implementación de la red neuronal artificial en el planificador . . . 38

6 Resultados experimentales 41
6.1 Análisis de los resultados . . . 41

7 Conclusiones 49
7.1 Visión general del trabajo realizado . . . 49

7.2 Relación con los estudios cursados 49
7.3 Trabajos futuros 50

Bibliografía **53**

Apéndices

A Código ejecución de los Benchmarks en lenguaje *C* **55**

B Código de la red neuronal artificial en lenguaje *Python* **73**

Índice de figuras

3.1 Ejemplo de la estructura de una jerarquía de memoria de tres niveles. . . . 12
3.2 *Deep learning* es solo una parte de la inteligencia artificial. 15
3.3 Una aproximación gráfica simple a una red neuronal artificial. 15

4.1 Arquitectura del procesador *IBM POWER8*. 20

5.1 IPC de cada una de las aplicaciones en ejecución individual empleando las configuraciones mencionadas anteriormente. 29
5.2 Espacio que representa la relación entre el evento *PM_L1_ICACHE_RELOADED_PREF* y las distintas configuraciones. 34
5.3 Resultados de la evaluación del modelo con los datos de test. 36

6.1 Prestaciones de las configuraciones estudiadas y nuestra implementación (*auto*) para los distintos *benchmarks*. 41
6.2 Rendimiento de las aplicaciones insensibles a la configuración del *prefetch*. 42
6.3 Historial de predicciones con respecto al rendimiento real de las aplicaciones insensibles a la configuración del *prefetch*. 43
6.4 Rendimiento de las aplicaciones sensibles a la configuración del *prefetch*. . 44
6.5 Porcentaje de predicción de las configuraciones del *prefetch* durante la ejecución de las aplicaciones. 45
6.6 Historial de predicciones con respecto al rendimiento real de las aplicaciones sensibles a la configuración del *prefetch*. 47

Índice de tablas

4.1 Estructura del registro DSCR. 21
4.2 Configuraciones del *prefetch* utilizadas. 22

5.1 Eventos utilizados como *inputs* para entrenar la red neuronal artificial que determinará la mejor configuración del *prefetch*. 31
5.2 Contadores a medir en cada uno de los tres intervalos virtuales. Solo puede haber cuatro contadores cada vez además de las instrucciones y los ciclos. 31

CAPÍTULO 1
Introducción

1.1 Descripción del problema

Actualmente existe un auge en el uso de algoritmos de *machine learning* para atacar problemas clásicos, con los que por falta de potencia en los computadores en la época en la que surgían, eran imposibles de implementar de forma eficiente. Con el avance de la tecnología, ha habido un incremento enorme en las prestaciones de los procesadores comerciales, con lo que estos algoritmos se vuelven más atractivos de implementar y utilizar, incluso en sistemas que requieren una respuesta muy rápida por parte de la red neuronal. Uno de los problemas clásicos más importantes, que esta causando más problemáticas los últimos años, es el de la configuración óptima del mecanismo de prebúsqueda.

Los procesos de fabricación de los procesadores han evolucionado de tal forma que permiten cada vez incorporar más transistores en el chip del procesador, como pronosticó la Ley de Moore. Esto ha permitido incrementar el tamaño y número de las memorias caché con el fin de reducir el tiempo medio de acceso a memoria, aprovechando los transistores adicionales incluidos en el chip. Las memorias caché se sitúan entre el procesador y la memoria principal, actuando de intermediarias para ocultar, al estar más cerca del procesador, la latencia de acceso a memoria. Su tamaño, en comparación con la memoria principal, permite que su tiempo de acceso sea mucho más reducido. Estas memorias permiten ocultar en gran medida la latencia de acceso a memoria principal gracias a la localidad que exhiben los datos e instrucciones. La diferencia de velocidad entre la memoria principal y las cachés hizo que los procesadores empezaran a organizar las cachés de manera jerárquica. Normalmente en los procesadores modernos se implementan tres niveles.

Sin embargo, utilizar una jerarquía de cachés no es suficiente para ocultar por completo la latencia de acceso a memoria, lo que puede puede repercutir negativamente en las prestaciones de muchas aplicaciones. Si los datos que necesita el procesador no se encuentran en ninguna de las memorias cachés, la latencia de acceso a los mismos se incrementa significativamente, ya que el tiempo de acceso a memoria principal suele ser superior al centenar de ciclos en los procesadores actuales, lo que conlleva una gran pérdida de prestaciones.

Estos accesos frecuentes a memoria principal suelen ser provocados por dos motivos. El primero viene dado por el incremento de potencia de los procesadores y aumento de su frecuencia de funcionamiento, mientras que en el caso de la evolución de las memorias caché no han habido aumentos tan significativos en su velocidad, sino más bien en sus tamaños, que cada vez son mayores. En segundo lugar, existe una limitación física

relacionada con los pines que se emplean para acceder al bus de memoria principal que acotan el ancho de banda disponible entre memoria y procesador.

Otra técnica que surge para minimizar la degradación de prestaciones provocada por los lentos accesos a memoria principal es la técnica de prebúsqueda hardware. Esta técnica se emplea para sacar el máximo beneficio de las jerarquías de caché. La prebúsqueda intenta prever qué datos o instrucciones van a ser usados por el procesador en los siguientes ciclos y trasladarlos a las memorias cachés de manera especulativa. Aunque como veremos más adelante en el presente trabajo, no siempre resulta beneficioso su uso o existen muchas configuraciones posibles que podrían mejorar su rendimiento pero es difícil identificar las configuraciones óptimas, poder configurar estos mecanismos de forma dinámica en ejecución para seleccionar la mejor configuración en función de las características de las aplicaciones resultaría una tarea muy interesante.

Este trabajo se enmarca dentro de un proyecto del plan de investigación estatal **"Tecnologías Innovadoras de Procesadores, Aceleradores y Redes para Centros de datos y Computación de Altas Prestaciones (T-PARCCA)"**. El objetivo final de este proyecto es aplicar técnicas de aprendizaje automático o *machine learning* (en inglés) a distintos mecanismos de la arquitectura del procesador, como la prebúsqueda hardware y la caché compartida. En este trabajo nos centramos en la configuración dinámica de la prebúsqueda hardware. En concreto, se presentan resultados para aplicaciones en ejecución individual en un solo núcleo que servirán de base para el desarrollo de la investigación posterior con un mayor número de núcleos y ejecuciones en paralelo.

1.2 Motivación

Desde el inicio de los primeros procesadores, el aumento de prestaciones en los procesadores ha inquietado a los arquitectos de computadores. A lo largo de los años, mientras los procesadores han ido evolucionando, su complejidad y prestaciones también lo han hecho. Hoy en día los procesadores están compuestos por múltiples núcleos en un mismo chip. Estos cuentan con sus recursos privados y además comparten otros, como son la la memoria principal y la memoria caché de último nivel (LLC), en los casos que la tengan.

Los procesadores modernos implementan mecanismos de prebúsqueda hardware o *prefetchers* para evitar esta pérdida de prestaciones. Para ello, predicen las instrucciones a ejecutar y los datos que utilizará el procesador y los traen de manera especulativa a la memoria caché. Como estos mecanismos son predicciones, pueden ser verdaderas o falsas, con lo que en algunos casos pueden mejorar el rendimiento de una aplicación, pero por el contrario también pueden empeorarlo.

Por ello, buscar una solución para poder configurar el *prefetcher* en tiempo real, obteniendo las mejores prestaciones para todo tipo de aplicaciones, según la influencia de la prebúsqueda en ella, puede resultar muy interesante.

Hoy en día existe un gran incremento de aplicaciones de las redes neuronales gracias a nuevas técnicas de aprendizaje profundo, un nuevo tipo de aprendizaje automático que ha sido posible gracias al incremento de la capacidad de cómputo y del uso de redes neuronales multicapa, las cuales actúan como aproximadores universales que permiten modelar relaciones no lineales entre datos de entrada y salida. Por esta razón, en este trabajo se estudiará la posibilidad de emplear un planificador dinámico para la configuración del *prefetcher* entrenado con algoritmos de *machine learning*.

1.3 Objetivos

En el presente trabajo se pretende utilizar técnicas de aprendizaje automático para implementar un programa que configure de manera dinámica, en tiempo de ejecución, la configuración del *prefetcher* adecuada para cada aplicación, con el fin de alcanzar las máximas prestaciones del sistema. Los objetivos planteados en este proyecto se pueden clasificar en:

1. Estudio y selección de las variables y parámetros necesarios para realizar el entrenamiento. El conjunto seleccionado debe proporcionar datos significativos que diferencien entre las configuraciones del *prefetcher* del sistema.
2. Diseño e implementación de un programa que obtenga los datos medidos en los contadores hardware y los normalice para realizar el entrenamiento de un modelo de *machine learning*.
3. Implementación de un algoritmo de *machine learning* capaz de clasificar las configuraciones del *prefetch* según su rendimiento.
4. Pruebas test y resultados teóricos para verificar el comportamiento del algoritmo de *machine learning*.
5. Desarrollo de un planificador dinámico que implemente el modelo de *machine learning* para configurar el *prefetcher*.
6. Evaluación del modelo entrenado en ejecución real.

1.4 Impacto esperado

El procesador *IBM POWER8* dispone de una configuración por defecto que alcanza buenas prestaciones para un rango amplio de aplicaciones, especialmente, cuando se ejecutan solas sobre el sistema. Este problema se agrava cuando existen múltiples aplicaciones en ejecución concurrente.

La configuración por defecto (U4P4) se aplica independientemente del tipo de aplicación que se encuentre en ejecución. Por el contrario, la técnica propuesta permite seleccionar la mejor configuración del *prefetcher* para cada aplicación en cada instante. En consecuencia, permite que el sistema en general consiga mejorar las prestaciones (IPC).

1.5 Metodología

Para para cumplir con los objetivos de este trabajo se han seguido los siguientes pasos:

1. Estudio sobre trabajos relacionados. Se ha hecho una búsqueda para conocer con detalle las implementaciones de modelos de *machine learning* para la configuración de la prebúsqueda actualmente en desarrollo o más recientes que resulten interesantes.
2. Estudio sobre *machine learning*. Se ha estudiado como implementar algoritmos *machine learning*, cuáles son los pasos para implementar un algoritmo de este tipo y conceptos previos a tener en cuenta.

3. Plataforma de trabajo. En este paso se ha estudiado la arquitectura del sistema dónde se realizarán las pruebas y las herramientas que se emplearán durante el desarrollo del mismo.

4. Análisis del problema. Este paso se ha desarrollado en las siguientes fases:

   a) Primero se ha estudiado el efecto que tienen las diversas configuraciones del *prefetch* en cada aplicación.

   b) En segundo lugar, se estudia el impacto de las diversas características que nos permitan obtener datos significativos para diferenciar entre las configuraciones del *prefetcher* que tenemos.

   c) Para finalizar esta fase, se desarrolla un programa que nos extrae y normaliza los datos de entrenamiento necesarios para entrenar un modelo de *machine learning*.

5. Implementación y entrenamiento de un algoritmo de *machine learning* capaz de clasificar los *prefetchers* según su rendimiento.

6. Implementación en la máquina real. En este paso se ha elaborado la propuesta de configuración dinámica, eligiendo la mejor configuración del *prefetcher* en función del rendimiento de la aplicación en tiempo real.

7. Validación de los resultados obtenidos. Último paso en este trabajo, en el que se valida el modelo en tiempo real mientras se ejecutan diversas aplicaciones para comprobar cuál es el rendimiento obtenido.

1.6 Estructura de la memoria

El presente trabajo de fin de grado se divide en los siguientes capítulos.

- Capítulo 2. Estado del arte. En este capítulo se resumen los principales trabajos científicos publicados recientemente relacionados con la temática del presente trabajo, diferenciando entre los trabajos relacionados con la jerarquía de memoria y el *prefetch*, y el uso de algoritmos de aprendizaje automático para la configuración de hardware.

- Capítulo 3. Conceptos básicos. En este capítulo se pretende introducir a los conceptos básicos que se deben tener en cuenta para entender el funcionamiento de los algoritmos de *machine learning*. Además se realiza una pequeña descripción del subsistema de memoria y los mecanismos de prebúsqueda, relevantes para entender el resto del trabajo.

- Capítulo 4. Entorno de trabajo. Se exponen las herramientas y materiales empleados en el desarrollo del presente trabajo. También se presenta la metodología usada para evaluar la propuesta.

- Capítulo 5. Configuración dinámica de prebúsqueda basada en *machine learning*. En este apartado se realiza el estudio de las diferentes aplicaciones para evaluar su comportamiento en términos de prestaciones empleando diversas configuraciones de *prefetch*. Se realiza un estudio de los mejores contadores de prestaciones disponibles para la posterior implementación de una red neuronal artificial. Se expone y explica como se ha desarrollado la propuesta.

- Capítulo 6. Resultados Experimentales. Se ejecutan pruebas y se evalúan los resultados.
- Capítulo 7. Conclusiones. Finalmente, presentaremos las conclusiones alcanzadas al terminar este trabajo, así como futuros posibles trabajos derivados de este e incluso publicaciones del mismo.
- Anexo A, código en *C* para ejecutar las aplicaciones con el planificador dinámico.
- Anexo B, código en *Python* empleado para ejecutar la red neuronal artificial.

1.7 Convenciones

En el desarrollo de la memoria para este trabajo se han seguido las siguientes convenciones:

- Mecanismo de prebúsqueda y prebúsqueda se utilizan de forma indistinta con *prefetcher* y *prefetch* respectivamente.
- Algoritmos de aprendizaje profundo y aprendizaje automático se usan de manera indistinta con algoritmos de *deep learning* o *machine learning*.
- Se han escrito en cursiva las siglas, nombres propios de aplicaciones, procesadores, algoritmos y palabras extranjeras.

CAPÍTULO 2
Estado del arte

En el presente capítulo se nombran los principales trabajos científicos publicados recientemente relacionados con la temática del presente trabajo, diferenciando entre los trabajos relacionados con la jerarquía de memoria y el *prefetch*, y el uso de algoritmos de aprendizaje automático para la configuración de hardware.

2.1 Trabajos relacionados con el *prefetch* y la jerarquía de memoria

La prebúsqueda de datos se ha investigado durante mucho tiempo y sigue siendo un tema de investigación activo, ya que puede afectar en gran medida el rendimiento de la jerarquía de memoria.

El estudio realizado en [6], propone *ADP*, que apaga el prefetcher en núcleos específicos cuando no se esperan beneficios locales o se está interfiriendo negativamente con otros núcleos. El componente clave de *ADP* es la política de activación que debe prever cuándo será beneficiosa la captación previa sin que el prefetcher esté activo. El prefetcher propuesto mejora tanto el rendimiento como el consumo energético.

En el trabajo realizado en [7], se propone *weighted-majority filter*, una forma experta de predecir la utilidad de las direcciones de memoria por parte de la prebúsqueda. El filtro propuesto es de naturaleza adaptativa y utiliza la predicción de los mejores predictores de un conjunto de predictores. Este filtro es ortogonal al algoritmo de captación previa subyacente.

Un trabajo anterior realizado en nuestra escuela [8], propone *Bandwidth-Aware Prefetcher Configuration* (BAPC), una estrategia de prebúsqueda que elige la mejor configuración para cada aplicación en cargas multiprogramadas ejecutándose en un sistema *IBM POWER8*. La propuesta se basa en la caracterización del comportamiento de las aplicaciones, en términos de prestaciones y consumo de ancho de banda. El estudio caracteriza las aplicaciones *prefetch friendly* en dos grandes grupos: *prefetch-configuration sensitive* y *prefetch-configuration insensitive*. La propuesta persigue aumentar el ancho de banda disponible eligiendo la configuración con menor demanda de este recurso para las aplicaciones *prefetch unfriendly* y *prefetch-configuration insensitive*.

2.2 Trabajos relacionados con la configuración de hardware empleando *machine learning*

El uso de algoritmos de aprendizaje automático para optimizar programas y sistemas se ha convertido en una tendencia popular para las comunidades de investigación de arquitectura, ya que se ha vuelto eficaz para resolver problemas no lineales, como la predicción y el ajuste del rendimiento.

En [9], los autores proponen una metodología basada en el aprendizaje automático para seleccionar cuál de los 4 *prefetchers* disponibles en los procesadores *Intel Core 2 Quad* debe activarse o desactivarse.

Otros trabajos recientes [10], propone un *framework* para la configuración dinámica del SMT (subprocesamiento múltiple simultáneo) basado en predicción (PBDST), para ajustar el recuento de subprocesos en los núcleos SMT de los procesadores *IBM POWER8* mediante el uso de algoritmos de aprendizaje automático. Su innovación radica en la adopción de predicciones de configuración SMT en línea derivadas del perfil de nivel de microarquitectura, para regular el recuento de hilos que podrían lograr un rendimiento casi óptimo.

El artículo [11] estudia el efecto de la prebúsqueda hardware en el código multiproceso y presenta una técnica de aprendizaje automático para predecir la combinación óptima de mecanismos de prebúsqueda para una aplicación determinada.

En [12] se estudian dos tipos de técnicas de prebúsqueda que están disponibles en un procesador *Intel Xeon Phi* de 61 núcleos, prebúsqueda software (guiada por compilador) y prebúsqueda hardware en una variedad de cargas de trabajo. Se emplean técnicas de aprendizaje automático, sintetizan las fases de la carga de trabajo y la secuencia de los patrones de fase utilizando datos de rendimiento de contadores de hardware, tales como el ancho de banda de memoria, índices de fallas, prebúsquedas previas emitidas, etc.

En la publicación [13], se demuestra el potencial del aprendizaje profundo para abordar el cuello de botella de *von Neumann*[1] en el rendimiento de la memoria. Se centran en aprender patrones de acceso a la memoria, con el objetivo de construir *prefetchers* de memoria precisos y eficientes. Relacionan las estrategias contemporáneas de prebúsqueda con los modelos de *n-gramas*[2] en el procesamiento del lenguaje natural, y demuestran cómo las redes neuronales recurrentes[3] pueden servir como un reemplazo directo.

Los autores del artículo [14], presentan como la localidad semántica puede capturar la relación entre los datos de una manera independiente del diseño de datos real, y argumentan que la localidad semántica trasciende en las preocupaciones espacio-temporales. Además, presentan el *context-based prefetcher*, que se aproxima a la localidad semántica mediante el aprendizaje por refuezo (redes neuronales recurrentes). El prefetcher identifica patrones de acceso a memoria mediante la aplicación de métodos de aprendizaje por refuerzo sobre la máquina y los atributos de código, que proporcionan pistas sobre la semántica de acceso a la memoria. Demuestran que, el *context-based prefetcher*, hace posible

[1]Idea de que el rendimiento del sistema informático es limitado debido a la capacidad relativa de los procesadores en comparación con las tasas máximas de transferencia de datos. Según esta descripción de la arquitectura de los computadores, un procesador está inactivo durante un cierto tiempo mientras se accede a la memoria.

[2]Tipo de modelo probabilístico que permite hacer una predicción estadística del próximo elemento de una secuencia de elementos sucedida hasta el momento.

[3]Área del aprendizaje automático inspirada en la psicología conductista, cuya ocupación es determinar qué acciones debe escoger un agente de software en un entorno dado con el fin de maximizar alguna noción de «recompensa» o premio acumulado.

que, implementaciones ingenuas basadas en punteros de algoritmos irregulares, logren un rendimiento comparable al del código espacialmente optimizado.

Por último en [15], presentan un esquema dinámico de ajuste de la prebúsqueda, denominado *Prefetch Automatic Tuner* (PATer). PATer utiliza un modelo de predicción basado en el aprendizaje automático para ajustar dinámicamente la configuración de prebúsqueda en función de los valores de los contadores que monitorizan el rendimiento del hardware. Demuestran que PATer es capaz de acelerar la ejecución de diversas cargas de trabajo hasta 1.4 veces.

CAPÍTULO 3
Conceptos básicos

En este capítulo se describen los conceptos básicos para facilitar la lectura y la comprensión del resto de capítulos del presente trabajo.

De manera análoga al capítulo anterior, los conceptos se encuentran agrupados según la temática que tratan: aquellos relacionados con los mecanismos de reducción de la latencia de acceso a memoria en general (y con las técnicas de prebúsqueda en particular), y aquellos relacionados con técnicas de *deep learning*.

3.1 Mecanismos de reducción de la latencia de acceso a memoria

3.1.1. Jerarquías de memoria

Uno de los mecanismos más utilizados hoy en día para ocultar la latencia de acceso a los datos e instrucciones que emplean los procesadores actuales a la hora de ejecutar aplicaciones son las jerarquías de memoria. Estas consiguen mejorar las prestaciones en la mayoría de los casos, ya que se componen de memorias de tamaños reducidos pero que son extremadamente rápidas. Los procesadores más modernos suelen disponer normalmente de tres niveles de caché organizados de manera jerárquica, como podemos observar en la Figura 3.1. Estas memorias presentan tiempos de acceso menores cuanto más pequeñas son. De esta forma, el primer nivel de caché, integrada en la estructura del procesador, es la más pequeña y rápida de la jerarquía. Siguiendo con esta afirmación, mientras avanzamos por esta jerarquía de memorias cada nivel implica un aumento de la capacidad de estas, sacrificando los tiempos de acceso, que aumentan en cada nivel a medida que se alejan del procesador.

La jerarquía de memoria presenta el funcionamiento que sigue. Cuando el procesador ejecuta una instrucción para acceder a datos en memoria, se hace una búsqueda por todos los niveles de la jerarquía empezando por la caché más cercana al procesador. Si se encuentran los datos en el primer nivel, caché L1 en la Figura 3.1, se produce un acierto (hit en inglés). Por otra lado, si no se encuentran los datos se produce un fallo (miss) en el nivel de caché en el que nos encontremos, y se generaría una petición de búsqueda del dato al nivel adyacente más alejado del procesador, si existe. En el caso de no existir un siguiente nivel, se produciría un acceso a memoria principal, lo que supondría un tiempo de acceso muy elevado que repercutiría negativamente en las prestaciones.

Este problema empeora en procesadores multinúcleo, dónde pueden haber múltiples aplicaciones compitiendo entre ellas por el bus de memoria para acceder a la memoria principal. Esto provoca en los procesadores actuales una perdida de prestaciones impor-

tante en la ejecución de aplicaciones, debido a la limitación física de los procesadores actuales para añadir más pines al chip del procesador. Es decir, el número de controladores de memoria que puede incorporar un procesador es limitado.

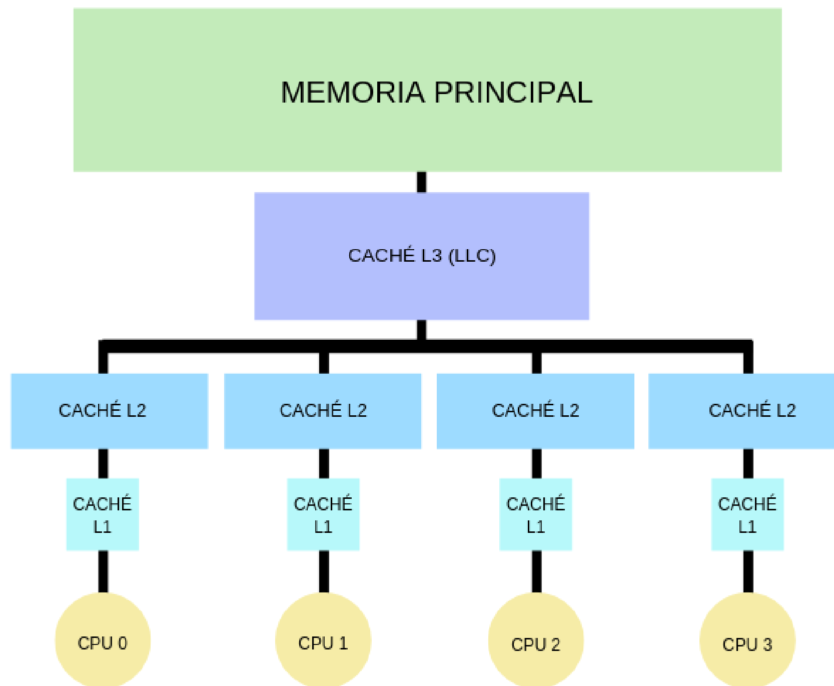


Figura 3.1: Ejemplo de la estructura de una jerarquía de memoria de tres niveles.

Para evitar las peticiones a memoria principal y conseguir buenas prestaciones, en los ultimos años se han aumentando las capacidades de las memorias cachés de los niveles más alejados del procesador, LLC. Estas cachés se componen de decenas de megabytes teniendo como objetivo aumentar la tasa de aciertos en las cachés, evitando en mayor medida el acceso a memoria principal. Normalmente las cachés de último nivel son compartidas por todos los núcleos del procesador para así tener un mayor aprovechamiento en el caso de que los núcleos vecinos no hagan uso de su espacio, pudiendo utilizarlo otro núcleo que sí lo necesite. A raíz de esto, surge el problema de que se generen interferencias entre núcleos y de esta forma haya un impacto negativo en las prestaciones de las aplicaciones que se esten ejecutando al mismo tiempo. En el caso del procesador empleado en el presente trabajo, el *IBM POWER8*, consta de 80MB empleando la tecnología *embedded DRAM*, eDRAM.

3.1.2. Prebúsqueda

El segundo mecanismo de ocultación de la latencia de acceso a memoria y que esta directamente relacionado con el anterior, es el mecanismo de prebúsqueda. Este mecanismo parte de la base de que la mayoría de las veces el uso de las jerarquias de memoria por sí solas no son suficientes para conseguir buenas prestaciones en la ejecución de algunas aplicaciones. A raíz de este problema, surgieron los mecanismos de prebúsqueda. Estos mecanismos intentan predecir qué instrucciones y/o datos va a utilizar el procesador en los siguientes ciclos de ejecución, y trata de traerlos cerca de él (a las cachés más cercanas)

antes de que éste los solicite. En otras palabras, los mecanismos de *prefetch* trabajan de forma paralela a las ejecuciones de las aplicaciones, que mientras están procesando unos datos, los mecanismos de *prefetch* están continuamente realizando peticiones a memoria de datos que posiblemente se vayan a utilizar en los siguientes ciclos y son llevados a las cachés para ocultar el tiempo de acceso a ésta. Esto conlleva, que si se produce un acierto (hit), habrá una reducción importante en la latencia de acceso a los datos. Por otra parte, realizar peticiones mediante especulaciones no siempre termina en aciertos, por lo que las peticiones pueden ser incorrectas o lanzarse demasiado tarde o pronto penalizando en el rendimiento.

Hay dos puntos clave que se deben cumplir en estos mecanismos para que presenten un buen funcionamiento, por un lado qué datos e instrucciones hay que predecir y en segundo lugar, en qué momento deben ser solicitados para que los datos lleguen en el momento idóneo [3].

Generalmente los *prefetchers* se clasifican en función de la tecnología empleada para su implementación. Diferenciamos entre dos tipos:

- **Los *prefetchers* software:** Se basan en la implementación de operaciones especiales mediante instrucciones que habitualmente son añadidas por los compiladores. Su objetivo principal es solicitar los datos que se esperan que van a ser necesarios en un momento futuro determinado y de esta forma llevarlos a las cachés. Para poder hacer esto de forma eficiente, el compilador debería conocer la estructura del procesador y el tamaño de las memorias.

  Como es necesario tener de instrucciones específicas en el código de las aplicaciones, se desarrollan compiladores determinados que son capaces de determinar la mejor posición para insertar las instrucciones de *prefetch*. Éstas, no son bloqueantes para el sistema en el caso de fallo (miss), sino que se ejecutan de manera concurrente con las instrucciones de las aplicaciones.

- **Los *prefetchers* hardware:** corresponden a lo *prefetchers* que se basan en una serie de estructuras hardware que son capaces de almacenar las actividades más recientes de las memorias. Hacen uso de hardware dedicado, incluído dentro del procesador, y su función es la de predecir qué datos e instrucciones hay que solicitar a la memoria y llevarlos directamente a las memorias caché. Gracias a la inclusión de este hardware, el cual no se encuentra en una ruta crítica, no contribuye a un incremento de los ciclos del procesador. Por otra parte, también tenemos inconvenientes a la hora de emplear este tipo de *prefetcher*, como son, la necesidad de disponer del hardware específico y el aumento en el consumo del ancho de banda.

Dado que el procesador *IBM POWER8* empleado en este trabajo dispone de hardware específico para el *prefetcher*, nos centraremos en este último tipo.

3.2 Machine learning

3.2.1. Definiciones previas

Antes de implementar una red neuronal artificial hay que entender qué son, como funcionan y la base de la que parten. Los conceptos que se describen a continuación han sido extraídos de libro de Jordi Torres [1]

El primer concepto que se debe tener claro para desarrollar una red neuronal artificial es *machine learning* o aprendizaje automático. Éste concepto se define como un subcampo

dentro de la inteligencia artificial, en otras palabras, permite a los computadores la capacidad de aprender sin ser programados directamente, para lograr una tarea se indican una serie de reglas que deben seguir, y estos son capaces de aprender de una forma automática. Generalizando, se podría decir que *machine learning* consiste en desarrollar para un problema en particular un algoritmo que sea capaz de predecir respuestas válidas en un escenario determinado. Éstos son capaces de aprender a partir de datos de muestra con el objetivo de encontrar tendencias o patrones, comprender la relación de estos datos, de tal forma que consiguen implementar un modelo capaz de predecir y clasificar elementos. Existen tres grupos principales según el tipo de entrenamiento que reciben para aprender:

- **Aprendizaje supervisado**: cuando los datos que se usan para el entrenamiento incluyen la solución esperada, denominada «etiqueta» (*label* en inglés). Los algoritmos que forman parte de este grupo y más conocidos son *la regresión lineal*, *la regresión logística*, *support vector machines*, *decision trees*, *random forest* y *redes neuronales*.
- **Aprendizaje no supervisado**: los datos de entrenamiento no incluyen la solución esperada (etiquetas), de tal forma que es tarea del algoritmo clasificar la información. Los algoritmos más populares de este grupo son *clustering (K-means)* o *principal component analysis (PCA)*.
- ***Reinforcement Learning* o aprendizaje por refuerzo**: cuando el modelo a implementar, debe explorar un entorno que desconoce y debe determinar que acciones llevar a cabo a modo de prueba y error. Este es capaz de aprender por si solo gracias a las penalizaciones y recompensas que recibe de las acciones que toma. Debe encontrar las mejores estrategias posibles con el fin de obtener una recompensa en tiempo y forma. Este tipo de aprendizaje se puede combinar con los demás tipos.

En segundo lugar, existe el concepto de *Artificial Neural Network* (ANN), o red neuronal artificial, que son sistemas informáticos que se inspiran en las redes neuronales biológicas que constituyen los cerebros, pero no son necesariamente idénticas. Estos sistemas aprenden a realizar tareas considerando ejemplos, generalmente sin ser programados con ninguna regla específica de tareas. Por ejemplo, en el reconocimiento de imágenes, pueden aprender a identificar imágenes que contienen perros analizando imágenes de ejemplo, que se han etiquetado manualmente como «perro» o «no perro», y usar los resultados para identificar perros en otras imágenes distintas. Lo hacen sin ningún conocimiento previo sobre los perros, por ejemplo, que tienen pelaje, colas, bigotes y caras de perros. En su lugar, generan automáticamente características de identificación a partir del material de aprendizaje que procesan.

El *deep learning* o aprendizaje profundo es un caso específico del machine learning (Figura 3.2) que emplea redes neuronales artificiales y se define como un algoritmo automático organizado de manera jerárquica o estructural que intenta imitar el aprendizaje humano con el objetivo de obtener unos determinados conocimientos. No requiere ser programado explícitamente, es el propio sistema el cuál es capaz de «aprender» por sí solo para realizar una tarea a través de una de un entrenamiento previo en dicha área. Así mismo, éstos constan de diversas capas de procesamiento para aprender representaciones de datos, con diversos niveles de abstracción que realizan una serie de transformaciones lineales y no lineales sobre estos datos, consiguiendo generar una salida similar a la esperada. En este caso se emplea un aprendizaje supervisado, que consiste en encontrar los parámetros de esas transformaciones (los pesos w_i y el sesgo b, que veremos en el siguiente punto), consiguiendo que éstas sean lo más óptimas, en otras palabras, la salida generada sea lo más similar a la esperada o que se distancien muy poco. Estos modelos están organizados en tres capas (Figura 3.3):

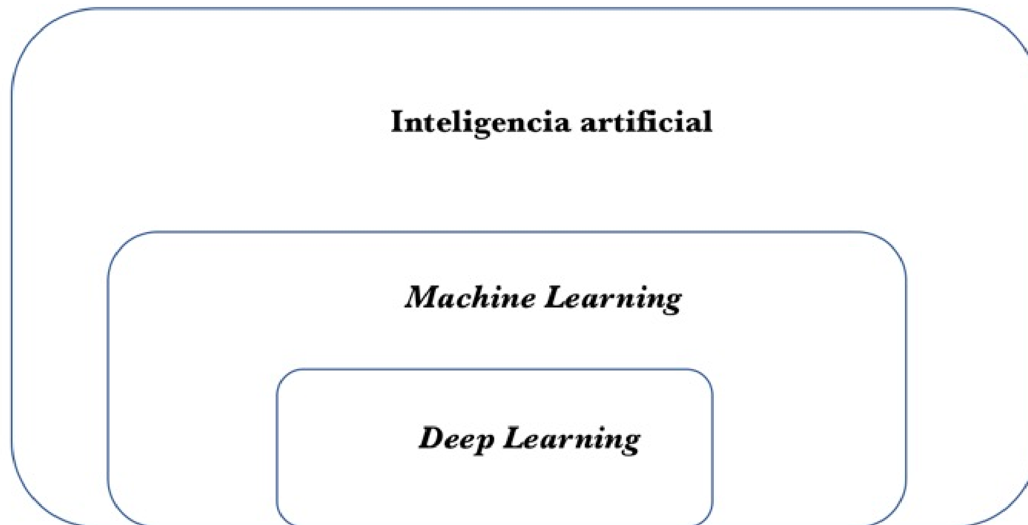


Figura 3.2: *Deep learning* es solo una parte de la inteligencia artificial.

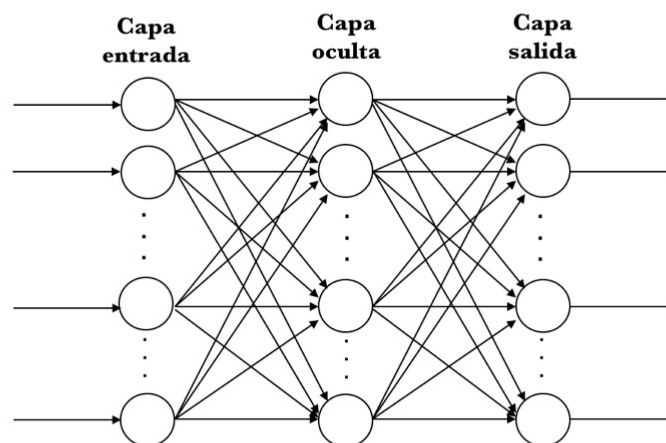


Figura 3.3: Una aproximación gráfica simple a una red neuronal artificial.

- **Capa de entrada (*Input Layer*)**: Compuesta por las neuronas que reciben los datos en la entrada, como podría ser un fichero con datos o una imagen .
- **Capa oculta (*Hidden Layer*)**: Consiste en la red que se encarga de realiza los cálculos internos y procesar la información. Puede haber muchas capas en este nivel y con distinsto número de neuronas en cada una de ellas. Además cuantas más neuronas haya en cada capa, más complejos serán dichos cálculos.
- **Salida (*Output Layer*)**: En última instancia se encuentra la capa de salida que se encarga de entregar cuál es el resultado de los cálculos anteriores.

3.2.2. Terminología básica

En este punto se va a describir la terminología básica de *machine learning*, en especial de los algoritmos de *deep learning*, que se emplearan en el desarrollo de la propuesta de este trabajo y que es necesaria conocer previamente al desarrollo de un algoritmo de este tipo.

En *machine learning* cuando se habla de *label* (o «etiqueta» en español) se refiere a lo que se está tratando de predecir con un modelo. En cambio, a una variable de entrada se la denomina *feature* («característica» o «variable» en español).

Un modelo define la relación entre *features* y *labels* y tiene dos fases claramente diferenciadas:

- **Fase de *training* (entrenamiento o aprendizaje)**: es cuando se «aprende» o se crea el modelo, a partir de los ejemplos de datos que le mostramos en la entrada y que tiene etiquetados, establece una relación entre ellos de tal forma que consigue aprender iterativamente las relaciones entre los *labels* y las *features* de los ejemplos.
- **Fase de *inference* (inferencia o predicción)**: Consiste en el proceso de realizar predicciones por parte del modelo entrenado anteriormente con ejemplos de datos no etiquetados.

Si consideramos el siguiente ejemplo, consiste en el modelo más simple que expresa la relación lineal entre *labels* y *features*. Éste se expresaría de la forma siguiente:

$$y = wx + b$$

En donde:

- ***y***: la etiqueta o *label* de una entrada de ejemplo.
- ***x***: la característica o *feature* del ejemplo de la entrada.
- ***w***: representa la pendiente de la recta peso o *weigth*, constituye uno de los parámetros que el modelo debe aprender durante el proceso de entrenamiento para emplearlo más tarde en el proceso de inferencia.
- ***b***: represente el punto de intersección de la recta en el eje sesgo o *bias*. Este es el otro de los parámetros que debe ser aprendido por el modelo.

En el modelo simple presentado anteriormente solamente existe una característica de entrada, en el caso de un algoritmo basado en *deep learning* se pueden presentar muchas variables de entrada, cada una con su peso w_i asociado. Como ejemplo, mencionaremos un modelo que emplea tres características (x_1, x_2, x_3), este podría formularse matemáticamente de la forma siguiente:

$$y = \sum_i w_i x_i + b$$

aquí se expresa la suma de los productos escalares entre los vectores X e Y, dónde más tarde se suma el sesgo.

Cuando se entrena un modelo, estos deben aprender los valores idóneos para los parámetros peso y sesgo. En el caso del aprendizaje supervisado, para conseguir dicho objetivo se aplican un algoritmo de aprendizaje automático capaces de obtener los valores para estos. Lo hacen examinando una gran cantidad de ejemplos que habremos etiquetado con la solución e intentará determinar los mejores valores para dichos parámetros minimizando el error entre la predicción y el valor real etiquetado (*loss*).

El *error* o *loss* es un concepto central en *deep learning* que representa la penalización de una mala predicción. Es decir, es un número que indica cuan mala ha sido una predicción

en un ejemplo concreto (si la predicción del modelo es perfecta, la *loss* es cero). Para determinar este valor, en el proceso de entrenamiento se usan las *funciones de loss*, que es una función matemática que agrega las *loss* individuales obtenidas de los ejemplos de entrada al modelo.

Finalmente, tenemos el concepto de *overfitting* (o «sobreajuste» en español) de un modelo, que se produciría en el caso de que el modelo se ajustara mucho a los ejemplos etiquetados en la entrada, imposibilitando las predicciones correctas por parte de nuestro modelo en ejemplos de datos nuevos que no habría visto antes.

CAPÍTULO 4
Entorno de trabajo

En el presente capítulo se describen el hardware y las herramientas utilizadas en el desarrollo de este trabajo.

4.1 Arquitectura del sistema

4.1.1. Procesador IBM POWER8

Para el desarrollo del presente trabajo se ha empleado un sistema *IBM Power System S812L*. Este sistema cuenta con un procesador *IBM POWER8*, y da vida a un sistema operativo Ubuntu 14.04[1] con el kernel de Linux 4.0.2. La arquitectura de este procesador puede llegar a soportar doce núcleos funcionando a una frecuencia de 4 GHz con soporte para ejecución simultánea (SMT) de hasta ocho hilos en cada uno de sus núcleos, obteniendo un total de 96 hilos de ejecución simultáneos. En la Figura 4.1 podemos ver cómo se organiza la arquitectura de este procesador.

En nuestro caso no contamos con la versión de doce núcleos, sino que tenemos una versión inferior del procesador que dispone únicamente de diez núcleos funcionando a 3,69 GHz con sólo un módulo de memoria DRAM de 32 GB, aunque comparte la misma arquitectura que la variante de doce núcleos. Como únicamente utiliza uno de los enlaces a memoria principal disponibles, existen núcleos más alejados a la memora principal que otros que están más cerca.

La jerarquía de memoria que ofrece el procesador *IBM POWER8* on-chip, está compuesta por tres niveles:

- **Primer nivel: L1D y L1I.** Este nivel se divide en dos cachés con propósitos diferentes. Una destinada al almacenamiento de datos y la otra para las instrucciones. Cuentan con capacidades de 64KB y 32KB respectivamente y son privadas, es decir, cada núcleo tiene sus propias memorias. Utilizan la tecnología *Static Random Access Memory* (SRAM).

- **Segundo nivel: L2.** Este nivel cuenta con un tamaño de 512KB y es privado en cada núcleo. Están construidas con la tecnología SRAM al igual que las cachés L1.

- **Tercer nivel: L3.** Este es el último nivel en la jerarquía de caché on-chip que posee este procesador y cuenta con un tamaño de 80 MB. En este caso, como en la mayoría de procesadores, al ser el último nivel de la jerarquía solo hay una memoria que es

[1]Sistema operativo de código abierto basado en GNU/Linux, que actualmente corre en computadores de escritorio y servidores, en arquitecturas Intel, AMD, POWER y ARM.

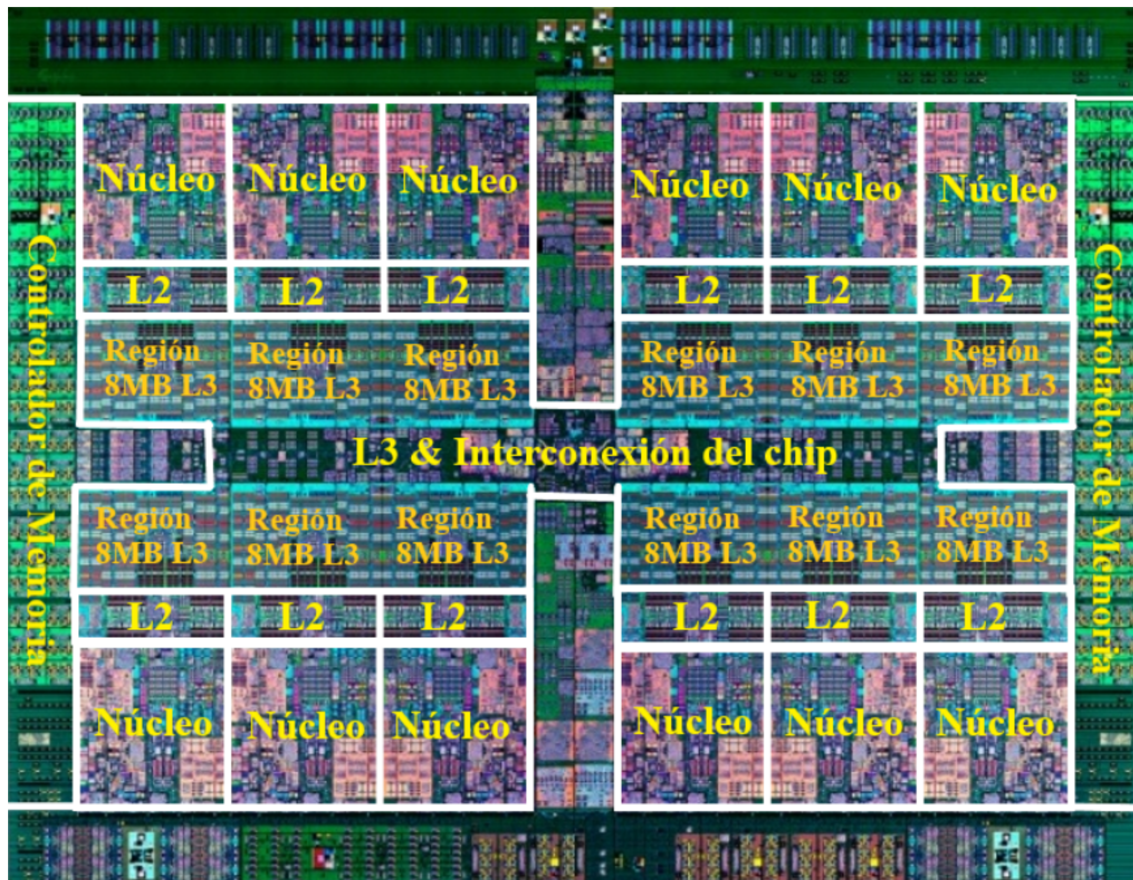


Figura 4.1: Arquitectura del procesador *IBM POWER8*.

compartida por todos los núcleos. La tecnología que usa es distinta a la empleada en las mencionadas anteriormente, en este caso utiliza memorias eDRAM, *Embedded dynamic random access memory*.

4.1.2. Mecanismos de prebúsqueda en el IBM POWER8

Prosiguiendo con el subsistema de memoria, el sistema del cual disponemos nos ofrece un sistema de prebúsqueda bastante complejo y sobradamente configurable. El *prefetch* de este procesador dispone de un registro destinado específicamente para la configuración del mismo, este se denomina *Data Streams Control Register*, DSCR. Esta formado por doce campos, como podemos observar en la Tabla 4.1 junto a los bits asociados y la característica que controlan. La complejidad de este *prefetcher* surge de la gran cantidad de configuraciones que existen del mismo, siendo exactos contamos con 2^{25} posibles configuraciones distintas, por lo que hacer un uso óptimo de él es una tarea extremadamente complicada.

Seguidamente se se explicarán detalladamente los campos presentados en la Tabla 4.1:

- ***Software Transient Enable (SWTE):*** Se aplica el atributo de transitorio a los flujos detectados por software.
- ***Hardware Transient Enable(HWTE):*** Se aplica el atributo de transitorio a los flujos detectados por hardware.

bit	Nombre del campo
0-38	
39	Software Transient Enable (SWTE)
40	Hardware Transient Enable (HWTE)
41	Store Transient Enable (STE)
42	Load Transient Enable (LTE)
43	Software Unit count Enable (SWUE)
44	Hardware Unit count Enable (HWUE)
45-54	Unit Count (UNITCNT)
55-57	Urgency (URG)
58	Load Stream Disable (LSD)
59	Stride-N Stream Enable (SNSE)
60	Store Stream Enable (SSE)
61-63	Default Prefetch Depth (DPFD)

Tabla 4.1: Estructura del registro DSCR.

- ***Store Transient Enable (STE):*** Se aplica el atributo de transitorio a los flujos de stores.
- ***Load Transient Enable (LTE):*** Se aplica el atributo de transitorio a los flujos de loads.
- ***Software Unit count Enable (SWUE):*** Aplica la cuenta de unidades a los flujos definidos por software.
- ***Hardware Unit count Enable (HWUE):*** Aplica la cuenta de unidades a los flujos definidos por hardware.
- ***Unit Count (UNITCNT):*** Número de unidades en un flujo de datos. Los flujos de datos que excedan este valor son finalizados.
- ***Depth Attainment Urgency (URG):*** Representa la velocidad o prioridad con la que las ráfagas de prebúsquedas llegan a la profundidad fijada. De manera equivalente al DPFD, el valor 0 selecciona la urgencia por defecto (urgencia 4). Así pues, la urgencia propiamente dicha varía desde 1 (sin urgencia) hasta 7 (máxima urgencia).
- ***Load Stream Disable (LSD):*** Este campo configura la detección e inicialización de ráfagas de loads (load streams).
- ***Stride-N Stream Enable (SNSE):*** Activa la detección hardware e inicialización de flujos de loads y stores que tienen una separación (stride) mayor que un solo bloque de cache. Estos flujos de loads pueden ser detectados cuando el campo LSD es 0, y los flujos de stores cuando el campo SSE se encuentra a 0 [4].
- ***Store Stream Enable (SSE):*** Activa la detección hardware e inicialización de flujos de stores.
- ***Default Prefetch Depth (DPFD):*** Representa la profundidad de la prebúsqueda en número de bloques. Como se ha comentado anteriormente, un valor 0 selecciona la profundidad por defecto (4 bloques). Por otro lado, el valor 1 indica profundidad nula o prebúsqueda desactivada (independientemente de otros campos de la configuración). Por lo tanto, la profundidad en número de bloques puede ser configurada entre los valores 2 (DPFD=010_2) y 7 (DPFD=111_2).

Configuración	Especificación	Valor
U1P2	URG=1 DPFD=2	66
U1P7	URG=1 DPFD=7	71
U7P2	URG=7 DPFD=2	450
U7P7	URG=7 DPFD=7	455

Tabla 4.2: Configuraciones del *prefetch* utilizadas.

Existen algunos casos especiales dentro de las muchas configuraciones posibles, tales como, cuando tenemos todos los campos de este registro a 0, dicho de otro modo DSCR=0, significa que el *prefetcher* se encuentra en la configuración por defecto. Dicha configuración es equivalente a que los campos de profundidad (DPFD) y urgencia (URG), configurados en el valor 4 para cada uno de ellos. Adicionalmente, estos no son los dos únicos campos se encuentran activos, sino que también existe un campo que al funcionar mediante lógica negativa (LSD), cuando almacena un 0 significa que la característica Load Stream está activada. El resto de campos quedarían desactivados en esta configuración.

Vamos a enfocar el presente trabajo en estudiar la posible viabilidad de una configuración dinámica del *prefetcher*, la cual mejore las prestaciones del sistema respecto a utilizar una única configuración durante toda la ejecución de una aplicación. En otras palabras, que sea capaz de identificar según el tipo de uso del *prefetch* que haga la aplicación durante su ejecución, cual de las posibles configuraciones del *prefetch* sería conveniente utilizar en cada instante de ejecución, cambiando entre ellas cuando sea conveniente. Por esto, se debe estudiar el comportamiento del sistema respecto a las distintas configuraciones posibles. Como hemos dicho antes, la cantidad de posibles configuraciones de este *prefetch* es extremadamente grande, por lo que en este trabajo hemos decidido reducir el análisis y nos hemos centrado en los campos que creemos que pueden resultar más influyentes, como la urgencia (URG) o la profundidad (DPFD). Los demás campos los mantendremos en sus configuraciones por defecto. En la Tabla 4.2 encontramos las configuraciones que emplearemos en el presente estudio.

4.1.3. Configuración del mecanismo de prebúsqueda

El registro mencionado anteriormente se puede modificar tanto para un identificador de proceso (*Process Identifier* (PID)) como para el sistema en general. En este trabajo se accede a él de la siguiente forma empleando un código escrito en lenguaje *C* (se expone un programa que posibilita el cambio del valor de dicho registro para un identificador de programa (*PID*) dado):

```
1  # define _GNU_SOURCE
2  # include <stdio.h>
3  # include <stdlib.h>
4  # include <stdint.h>
5  # include <unistd.h>
6  # include <string.h>
7  # include <sys/ptrace.h>
8  # include <errno.h>
9  # define PTRACE_DSCR 44
10 
11 static int do_dscr_pid(int dscr_state, pid_t pid)
12 {
13     int rc ;
```

```
14     rc = ptrace (PTRACE_ATTACH, pid , NULL , NULL);
15     if(rc) {
16         fprintf(stderr, "Could not attach to process %d to %s the "
17             " DSCR value \n%s\n", pid, (dscr_state ? "set" : "get"),
18             strerror(errno));
19         return rc;
20     }
21     wait(NULL);
22     rc = ptrace(PTRACE_POKEUSER, pid, PTRACE_DSCR << 3, dscr_state);
23     if(rc) {
24         fprintf(stderr, "Could not set the DSCR value for pid "
25             " %d\n%s\n", pid, strerror(errno));
26         ptrace(PTRACE_DETACH, pid, NULL, NULL);
27         return rc;
28     }
29     rc = ptrace(PTRACE_PEEKUSER, pid, PTRACE_DSCR << 3, NULL);
30     if(errno){
31         fprintf(stderr, "Could not get the DSCR value for pid "
32             " %d\n%s\n", pid, strerror(errno));
33         rc = −1;
34     } else {
35         printf("DSCR for pid %d is %d\n", pid, rc);
36     }
37     ptrace(PTRACE_DETACH, pid, NULL, NULL);
38     return rc;
39 }
40 int main(int argc, char *argv[])
41 {
42     pid_t pid = getpid(); //PID del que se tiene que cambiar
43     int dscr_val; //Valor del dscr
44     printf("Executant...\nTinc el pid: %d .\n", pid);
45     printf("Introdueix l'objectiu pid: ");
46     scanf("%d" ,&pid);
47     printf("Introdueix el valor de DSRC: ");
48     scanf("%d" ,&dscr_val);
49     do_dscr_pid(dscr_val, pid);
50     sleep(1);
51     printf("Acabant...\n");
52 }
```

Para poder utilizar este código se precisa del uso de una función llamada *ptrace*, ésta permite modificar el registro DCSR para cada uno de los procesos que se estén ejecutándo. Dicha función, se describe de la siguiente manera,

```
long ptrace(enum __ptrace_request request, pid_t pid, void *addr, void *data)
```

y permite a un proceso (observador, *tracer*), poder supervisar y controlar la ejecución de otro proceso (observado, *tracee*). Esto nos permite cambiar y examinar la memoria y registros usados por el proceso observado.

Como es necesario emplear la función descrita, hay que asociar los procesos. En las líneas 13-20 se emplea la solicitud *PTRACE_ATTACH*, que permite empezar el con el control del proceso indicado a partir se su *pid* (identificador de proceso). Una vez asociados los procesos, en las líneas de la 22 a la 28, se realiza una solicitud *PTRACE_POKEUSER*, que permite acceder al desplazamiento dentro de la pila del proceso que se observa y de este modo poder cambiar el valor por el que indicamos por parámetro. Más tarde, en las líneas 29-36, se comprueba que el valor se escribió correctamente empleando la función *PTRACE_PEEKUSER*, permitiendo leer parte de la pila en la memoria del proceso que se observa. Por último, en la línea 37, se libera el proceso que se estaba observando

con *PTRACE_DETACH*, desvinculándolo evitando cualquier tipo de error que pudiese generarse en caso de no hacerlo.

Al final de todo, de las líneas 40 a 52, se expone un pequeño programa de consola para que de manera intuitiva y sencilla, podamos modificar el valor del registro del proceso que indiquemos.

4.2 Contadores de prestaciones

Una de las tareas más complejas, aunque importante en los sistemas actuales, es la evaluación de sus prestaciones, debido a que su rendimiento y capacidad de cómputo han sido incrementadas junto con la complejidad de su configuración. Por esto, uno de los temas claves para observar las prestaciones que se obtienen en los sistemas son poder monitorizar el comportamiento de estos, para de esta forma, evaluar la efectividad de las configuraciones o mejoras añadidas.

Esta información de las prestaciones del comportamiento de un sistema se pueden obtener de diversas formas. En primer lugar, mediante el uso de instrucciones dentro del código fuente o en opciones dentro del compilador, lo que conlleva a modificar y aumentar el tamaño del código a evaluar tanto en complejidad como en tamaño. En segundo lugar, se podría medir eventos del sistema empleando contadores de prestaciones, los cuales hacen uso de un hardware dedicado dentro del procesador para leer registros destinados a almacenar la información sobre el comportamiento de los sistemas en tiempo de ejecución. Utilizando estos contadores hardware, no precisamos de modificar el código que queremos analizar. Además, son dependientes por completo de la arquitectura del sistema en el que se vayan a usar, por lo que se debe prestar especial atención a los eventos que tengamos disponibles para monitorizar en el sistema que queramos realizar las evaluaciones. Concretamente, el procesador elegido para nuestras pruebas, *IBM POWER8*, dispone de seis registros dedicados a estos contadores, donde solamente cuatro de ellos pueden ser configurados para monitorizar un evento diferente. Los dos contadores restantes son estáticos y sirven para medir las instrucciones ejecutadas y los ciclos transcurridos. Existen muchísimos eventos que pueden medir infinidad de eventos interesantes, como fallos de caché o fallos de predicción del *prefetcher*, entre muchos otros.

Existen muchas herramientas para medir estos eventos. En el caso particular de este estudio se ha usado de la herramienta perf[2] , que forma parte del kernel del sistema Linux desde la versión 2.6.31, haciendo uso de la librería *libpfm*[3]. Ésta nos facilita la configuración de los contadores de prestaciones mediante una interfaz genérica, a partir de una lista de eventos. Además, posee traducciones literales de los nombres de los eventos que podemos configurar en los contadores del procesador.

Los eventos monitorizados en este estudio son los que siguen (en el capítulo 5 se comentará la importancia de cada uno de ellos para este trabajo):

- ***cycles*:** Número de ciclos usados en la ejecución de la aplicación.
- **textitinstructions:** Número de instrucciones ejecutadas por la aplicación.
- ***PM_L1_ICACHE_MISS:*** Demandas de instrucciones a caché L1 fallidas.
- ***PERF_COUNT_HW_CACHE_L1D:READ:MISS:*** Fallo de lectura en caché L1D debido al *prefetch*.

[2]http://man7.org/linux/man-pages/man1/perf.1.html
[3]http://perfmon2.sourceforge.net/

- ***PERF_COUNT_HW_CACHE_L1D:WRITE:MISS:*** Fallo de escritura en caché L1D debido al *prefetch*.
- ***PERF_COUNT_HW_CACHE_LL:WRITE:MISS:*** Fallo de escritura en el último nivel de caché debido al *prefetch*.
- ***PM_DATA_FROM_L3:*** La memoria caché de datos del procesador se volvió a cargar desde la L3 del núcleo local debido a cargas de demanda.
- ***PM_MEM_PREF:*** Número de accesos a la memoria principal realizados por el *prefetcher* en la ejecución de la aplicación.
- ***PERF_COUNT_HW_CACHE_L1I:PREFETCH:ACCESS:*** Accesos del *prefetch* a lamemoria L1I.
- ***PERF_COUNT_HW_CACHE_L1D:PREFETCH:ACCESS:*** Accesos del *prefetch* a lamemoria L1D.
- ***PM_MEM_READ:*** Lecturas a memoria principal.
- ***PM_L1_ICACHE_RELOADED_PREF:*** Cuenta todas las recargas debidas al prefeth de la caché L1 de instrucciones.
- ***PM_DATA_FROM_MEMORY:*** La memoria caché de datos del procesador se volvió a cargar desde una ubicación de memoria que incluye L4 debido a las cargas de demanda.
- ***PM_BR_MPRED_CMPL:*** Número de predicciones del *prefetch* erróneas.

Para evaluar el rendimiento de las aplicaciones hemos utilizado la siguiente métrica, que nos indica cuantas instrucciones es capaz de ejecutar el procesador en un ciclo de reloj:

- **Instruciones por ciclo (IPC)**: Número de instrucciones ejecutadas por ciclo de la aplicación.

$$IPC = \frac{Instrucciones\ ejecutadas\ (instructions)}{Ciclos\ usados\ (cycles)}$$

4.3 *Benchmarks* SPEC CPU 2006

En el desarrollo del presente trabajo se han empleado algunos *benchmarks* de la suite de *Standard Performance Evaluation Corporation (SPEC) CPU 2006*[4] empleando las entradas *reference*. Estos *benchmarks* nos ayudarán a comparar nuestras propuestas o cambios, ya que nos proporcionan una medida estándar o comparable al resto de computadores. Para poder evaluar con el mismo peso todas las aplicaciones de la suite, se ha decidido ejecutar de forma individual, con cada una de las configuraciones del *prefetch* preseleccionadas (Tabla 4.2), durante 120 segundos. Cuando se han finalizado estas ejecuciones, se ha medido el rendimiento aplicando la métrica anterior y se ha almacenado para su posterior uso.

Se ha empleado esta suite de aplicaciones con el objetivo de obtener una medida comparativa de la intensidad de computación en el amplio rango de opciones del hardware,

[4]https://www.spec.org/cpu2006/

utilizando como base de la aplicación cargas de trabajo desarrolladas por usuarios representativos.

Estas aplicaciones incluidas en el paquete *SPEC CPU2006* [5] usadas en este trabajo, se dividen en dos grupos según el tipo de cálculos que realizan:

- **Aplicaciones de números enteros:** *bzip2, gcc, mcf, hmmer, sjeng, libquantum, h264ref, omnetpp, astar y xalancbmk.*
- **Aplicaciones de coma flotante:** *bwaves, gamess, milc, zeusmp,gromacs, cactusAMD, leslie3D, namd, soplex, povray y GemsFDTD.*

4.4 Herramientas

Para la realización de este trabajo se han usado las siguientes herramientas, además de las mencionadas anteriormente en este capítulo:

- ***Shell scripts:*** Se han utilizado estos archivos para agilizar y automatizar el lanzamiento de las aplicaciones mediante comandos, se escriben todos en un mismo fichero y automáticamente se ejecutan unos detrás de otros.

```
{ ./startExperiments_Script.sh; }

for workload in 1 2 3 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 21 22 23
do
  for dscr in 0 1 66 71 450 455
  do
    ./PlanificadorObtenerDatos -d 200 -v -e cycles,instructions,PM_L1_ICACHE_MISS,PERF_COUNT_HW_CACHE_L1D:READ:MISS,PERF_COUNT_HW_CACHE_L1D:WRITE:MISS,PERF_COUNT_HW_CACHE_LL:WRITE:MISS -q -A 0 -B $workload -C $dscr 2>> /home/malursem/working_dir/ManelOuts/experimento/trabajo[${workload}]conf[${dscr}].txt
  done;
done;

{ ./endExperiments_Script.sh; }
```

En el ejemplo de arriba, vemos un script que utilizamos para lanzar varias aplicaciones (bucle *for workload*) con diversas configuraciones (bucle *for dscr*), pudiendo guardar los resultados por separado, archivos *trabajo[${workload}]conf[${dscr}].txt*, donde *${workload}* toma el valor de la aplicación que se va a ejecutar y *${dscr}* toma el valor de la configuración del *prefetch* para esa ejecución. Como se puede observar esto facilita mucho la obtención de datos y su organización de manera automática.

Adicionalmente estos ficheros pueden ejecutar otros shell scripts. Como vemos en el código anterior, existen dos scripts al principio y al final del fichero (l. 1 y l. 11 respectivamente) que son importantes para nuestro trabajo ya que el contenido de dichos scripts hace que la máquina trabaje siempre a la misma frecuencia durante todas las pruebas, con el fin de obtener resultados equiparables.

```
for cpu in $(seq 1 79);
do
  { cpufreq-set -g userspace -c $cpu; }
done;

{ cpupower frequency-set -f 3690000; }

echo "CPU Ready..."
```

- **Programas escritos en lenguaje *C*:** En este trabajo se han empleado programas escritos en lenguaje *C* para lanzar los *benchmarks*, así como también poder medir los eventos configurando los contadores de prestaciones mediante la librería *libpfm*, que hemos mencionado anteriormente, y controlar que su ejecución dure el número de instrucciones correspondiente. Para la ejecución de mezclas se usa un código que también se encarga de controlar el valor de la configuración del *prefetcher* para cada una de las aplicaciones que la componen. Se ha partido de la infraestructura software base desarrollada por Josué Feliu[5] durante su tesis doctoral. Este software se ha modificado lo suficiente para habilitar el acceso a diferentes contadores de prestaciones, así como hacer posible el acceso a los registros de configuración del *prefetcher* individual de cada aplicación y permitir la implementación de la propuesta, habilitando adicionalmente, a ejecutar código escrito en *Python* para llamar a la red neuronal artificial (Anexo B). La versión de código utilizada para el lanzamiento de la propuesta está disponible en el Anexo A. En este código se hace uso de la librería *libpfm*, además de la función de cambio de DSCR por código que se mostraba anteriormente.

- **Programas escritos en lenguaje *Python*[6]:** Este lenguaje de programación permite manejar datos de forma muy fácil y rápida, por lo que se ha elegido para implementar programas de procesamiento de datos para su posterior uso en la generación de gráficos, con otras herramientas que mencionaremos más adelante.

  Por otra parte, este lenguaje es compatible con la *API* de *machine lerning* que hemos utilizado para implementar nuestra red neuronal artificial, descrita en el siguiente punto.

- ***API BigML*[7]:** Esta *API* permite realizar en su plataforma online todo el análisis e implementaciones necesarias para resolver un problema de *machine learning*. Ofrece un amplio abanico de algoritmos tanto supervisados como no supervisados, herramientas para normalización de datos, herramientas para generar nuevos datos, herramientas de entrenamiento y testeo de resultados, almacenamiento en la nube, y lo más importante, procesamiento de datos dedicado para entrenar nuestra red neuronal artificial. Esta *API* es compatible con algunos lenguajes de programación como *Python*, *Node.js*, *Ruby*, *Java*, *Swift* y más.

- **Aplicaciones de ofimática *Microsoft Excel*[8] *y Libre Office*[9]:** Se han utilizado únicamente para la generación de gráficos a partir de los datos obtenidos.

[5]Josué Feliu es el director experimental del presente trabajo.
[6]https://www.python.org/
[7]https://bigml.com/
[8]https://products.office.com/es-es/excel
[9]https://es.libreoffice.org/

CAPÍTULO 5

Configuración dinámica de prebúsqueda basada en machine learning

5.1 Efectos de las configuraciones del *prefetcher* en cada aplicación

En este capítulo se estudian los efectos que pueden tener las distintas configuraciones del *prefetcher* utilizando como referencia los resultados de los *benchmarks* mencionados anteriormente (*SPEC CPU 2006*) durante la ejecución individual de los mismos. Para estas pruebas se ha ejecutado cada *benchmark* con las seis configuraciones que se han comentado en capítulos anteriores. Las configuraciones que se emplean en estas pruebas son las configuraciones por defecto (*DEF* (DSCR=0)) y no *prefetch* (*OFF*, prebúsqueda desactivada (DPFD=1)), así como cuatro configuraciones adicionales variando la urgencia (*URG*) y la profundidad (*DPFD*) en sus valores máximos y mínimos. Concretamente estas son *U1P2, U1P7, U7P2 y U7P7*, donde *UxPy* corresponde a urgencia (*URG*) x y profundidad (*DPFD*) y.

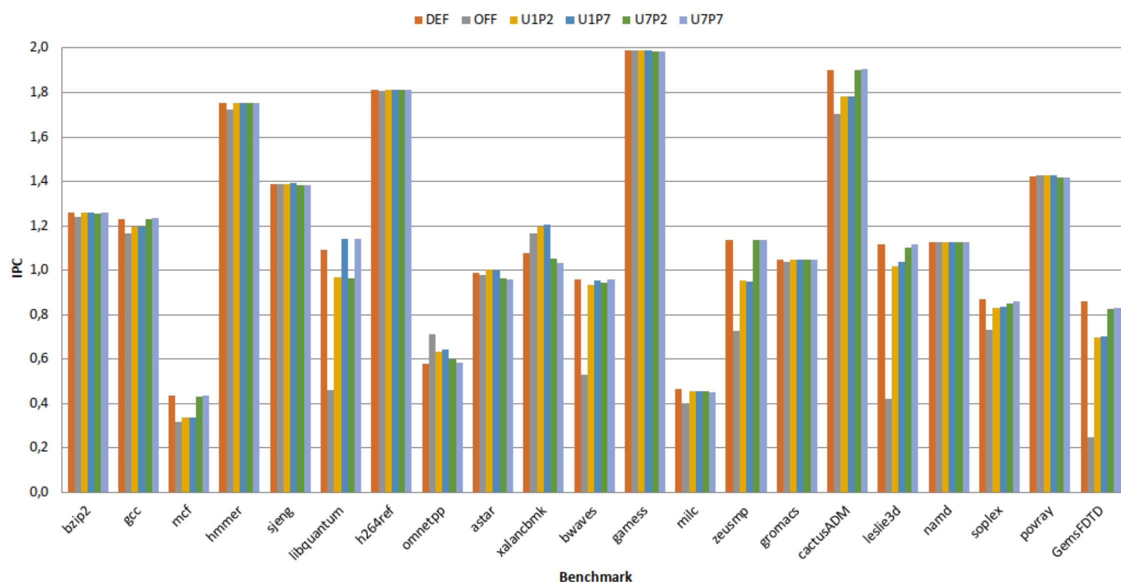


Figura 5.1: IPC de cada una de las aplicaciones en ejecución individual empleando las configuraciones mencionadas anteriormente.

En la Figura 5.1 se observa la influencia de las configuraciones del *prefetcher* en cada aplicación, cuando estas se ejecutan individualmente. De estos resultados obtenemos las siguientes observaciones importantes para nuestro estudio:

- En primer lugar se aprecia que empleando la configuración del *prefetch* por defecto (*DEF*), que dispone este sistema, superar sus resultados en muchos casos es bastante complicado.
- En segundo lugar, se observa que no todas las aplicaciones se benefician del mecanismo de *prefetch*, hay aplicaciones dónde la configuración no aporta beneficios en el rendimiento, ya sea con *prefetch* apagado o con cualquiera de las configuraciones que se han empleado en el estudio, como por ejemplo *gamess*, *h264ref* o *povray*. Sin embargo, existen casos en los que sí existen mejoras, algunas bastante significativas, como *GemsFDTD*, *leslie3D*, *cactusADM* o *bwaves*. Esto indica que habrá casos en los que es mejor apagar el *prefetch* para ahorrar ancho de banda, ya que no beneficiaría en el rendimiento, y otros casos en los que por contra, encenderlo mejora significativamente el rendimiento.
- Por último, se puede apreciar que la mayoría de las aplicaciones se benefician más de la profundidad, como *libquantum* o *zeusmap*. Por otra parte existen otras, aunque son menos, que por el contrario se benefician más de la urgencia, como por ejemplo *gcc* o *mcf*.

A la vista de los resultados, se concluye que para planificar dinámicamente el *prefetcher* hay que tener en cuenta estos tres casos para que la red neuronal artificial sea capaz de predecir la mejor configuración en todos los casos vistos en estas pruebas, y de esta manera no perder prestaciones, o activarlo cuando realmente no hace falta. Si se procura mantener estos casos cubiertos en nuestra implementación, las prestaciones del planificador dinámico serán las más adecuadas.

5.2 Diseño de la propuesta

5.2.1. Obtención de los datos

Como se describió en la sección 4.2, el *IBM POWER8* implementa seis contadores de eventos hardware que pueden configurarse para monitorizar diferentes eventos en tiempo de ejecución. Estos contadores pueden brindar información útil del rendimiento del sistema, como por ejemplo, el número de accesos y fallos en cada nivel de la jerarquía de memoria. La información obtenida a través de estos contadores es muy útil para saber como se comporta nuestro procesador mientras ejecuta las aplicaciones. Sabiendo esto, se ha querido aprovechar la información de dichos contadores para entrenar una red neuronal artificial y que, en tiempo de ejecución, sea capaz de elegir la mejor configuración del *prefetch* en función del comportamiento de la aplicación.

El primer paso para diseñar la red neuronal artificial consiste en analizar que características o *inputs* (en nuestro caso eventos monitorizables con los contadores de prestaciones) existen para diferenciar las distintas posibilidades del *output* de esta red neuronal artificial, es decir, la mejor configuración del *prefetch*. De todos los eventos disponibles en el *IBM POWER8*[1] resultan especialmente interesantes aquellos relacionados con la jerarquía de memoria y el *prefetch* (Tabla 5.1). Así pues, todos los eventos elegidos están relacionan con algún nivel de la jerarquía de memoria del procesador (descrita en el capítulo 4 sección 4.1), o en el mecanismos de prebúsqueda.

[1]http://oprofile.sourceforge.net/docs/ppc64-power8-events.php

Jerarquía de memoria	Prefetch
PM_L1_ICACHE_MISS	PERF_COUNT_HW_CACHE_L1D:READ:MISS
PM_DATA_FROM_L3	PERF_COUNT_HW_CACHE_L1D:WRITE:MISS
PM_MEM_READ	PERF_COUNT_HW_CACHE_LL:WRITE:MISS
PM_DATA_FROM_MEMORY	PM_MEM_PREF
	PERF_COUNT_HW_CACHE_L1I:PREFETCH:ACCESS
	PERF_COUNT_HW_CACHE_L1D:PREFETCH:ACCESS
	PM_L1_ICACHE_RELOADED_PREF
	PM_BR_MPRED_CMPL

Tabla 5.1: Eventos utilizados como *inputs* para entrenar la red neuronal artificial que determinará la mejor configuración del *prefetch*.

Intervalo 0
cycles
instructions
PM_L1_ICACHE_MISS
PERF_COUNT_HW_CACHE_L1D:READ:MISS
PERF_COUNT_HW_CACHE_L1D:WRITE:MISS
PERF_COUNT_HW_CACHE_LL:WRITE:MISS
Intervalo 1
cycles
instructions
PM_DATA_FROM_L3
PM_MEM_PREF
PERF_COUNT_HW_CACHE_L1I:PREFETCH:ACCESS
PERF_COUNT_HW_CACHE_L1D:PREFETCH:ACCESS
Intervalo 2
cycles
instructions
PM_MEM_READ
PM_L1_ICACHE_RELOADED_PREF
PM_DATA_FROM_MEMORY
PM_BR_MPRED_CMPL

Tabla 5.2: Contadores a medir en cada uno de los tres intervalos virtuales. Solo puede haber cuatro contadores cada vez además de las instrucciones y los ciclos.

Para construir una red neuronal artificial se precisa de grandes cantidades de datos relevantes para poder entrenar el modelo. Para obtener estos datos, se ha ejecutado las múltiples aplicaciones de la suite *SPEC CPU2006* configurando contadores de prestaciones del procesador para monitorizar los eventos seleccionados anteriormente. Estos eventos se monitorizan en intervalos de 200 ms durante la ejecución de cada aplicación. Como el procesador *IBM POWER8* solamente posee cuatro contadores de prestaciones configurables (los dos contadores restantes únicamente monitorizan el número de ciclos de ejecución y las instrucciones ejecutadas) y se necesita monitorizar más de cuatro características para obtener información suficiente con la que entrenar nuestro modelo, se emplearán intervalos virtuales. Es decir, para un intervalo real tendrán que transcurrir tres intervalos virtuales en los cuales se irá intercambiando entre los distintos eventos, Tabla 5.2. Para ello, se emplea un programa escrito en lenguaje C que planifica cada aplicación a ejecutar el número de instrucciones determinadas (para cada aplicación se ejecuta el número de instrucciones necesario para que la ejecución dure 120 segundos), y de esta

forma se obtienen los valores de los contadores pertinentes en cada uno de los intervalos. Así, tras varias ejecuciones de los distintos *benchmarks* con las distintas configuraciones del *prefetch*, obtendremos datos suficientes para entrenar la red neuronal artificial.

```
int measure() {
  int i, ret, errorPred = 1;
  // Libera los procesos
  for (i=0; i<N; i++) {
    if (queue[i].pid > 0) {
      kill(queue[i].pid, 18); //Reanuda Procesos
    }
  }
  // Mira si alguno ha fallado
  for (i=0; i<N; i++) {
    waitpid(queue[i].pid, &(queue[i].status), WCONTINUED);
    if (WIFEXITED(queue[i].status)) {
      //fprintf(stderr, "ERROR: command process %d_%d exited too early with
          status %d\n", queue[i].benchmark, queue[i].pid, WEXITSTATUS(queue[i].
          status));
    }
  }
  // Ejecuta 1 quantum
  usleep(options.delay*1000);
  // Bloquea los procesos
  ret = 0;
  for (i=0; i<N; i++) {
    if (queue[i].pid > 0) {
      kill(queue[i].pid, 19);
      //    waitpid(aux->pid, &(aux->status), WUNTRACED);
    }
  }
  // Mira si alguno ha fallado
  for (i=0; i<N; i++) {
    waitpid(queue[i].pid, &(queue[i].status), WUNTRACED);
    if (WIFEXITED(queue[i].status)) {
      //fprintf(stderr, "Process %d_%d finished with status %d\n", queue[i].
          benchmark, queue[i].pid, WEXITSTATUS(queue[i].status));
      ret++;
      queue[i].pid = -1;
    }
  }
  // Lectura de contadores
  for (i=0; i<N; i++) {
    get_countsv2(&(queue[i]));
    queue[i].instruccionesTotales += queue[i].counters[1] ; //Sumaho a la
        variable general, si iguales soles son els del quantum
  }
  //calculamos el quantum virtual de la siguiente iteracion
  if(virtualCount < virtualQuantums){
    virtualCount++;
  }else{
    planificar = 1;
    virtualCount = 0;
  }
  for (i=0; i<N; i++) {
    //finalitzar events
    finalitzar_events(&(queue[i]));
    //cambiar contadors
    if(i==0){
      switch (virtualCount)
      {
        case 0:
```

```
55          options.events=strdup("cycles,instructions,PM_L1_ICACHE_MISS,
                PERF_COUNT_HW_CACHE_L1D:READ:MISS,PERF_COUNT_HW_CACHE_L1D:WRITE:
                MISS,PERF_COUNT_HW_CACHE_LL:WRITE:MISS");
56          break;
57        case 1:
58          options.events=strdup("cycles,instructions,PM_DATA_FROM_L3,
                PM_MEM_PREF,PERF_COUNT_HW_CACHE_L1I:PREFETCH:ACCESS,
                PERF_COUNT_HW_CACHE_L1D:PREFETCH:ACCESS");
59          break;
60        case 2:
61          options.events=strdup("cycles,instructions,PM_MEM_READ,
                PM_L1_ICACHE_RELOADED_PREF,PM_DATA_FROM_MEMORY,PM_BR_MPRED_CMPL")
                ;
62          break;
63        default:
64          break;
65      }
66    }
67    // Iniciar contadores
68    iniciar_contadors (&(queue[i]));
69    //##### INICIAR EVENTOS Y CONTADORES SI PID != -1.
70    //SI PID==-1 SE DEBE RELANZAR SIEMPRE QUE NO SE HAYA PASADO LAS
          INSTRUCCIONES ( llansar_proces(&queue[i]) )
71    //y luego iniciar contadores y eventos
72    if(queue[i].pid == -1){
73      if(queue[i].instruccionesTotales < instruccions_totals[queue[i].benchmark
            ] && !queue[i].finished){
74        llansar_proces(&queue[i]);
75        iniciar_events(&(queue[i]));
76      }else{
77        end_experiment++;
78        queue[i].finished = 1;
79      }
80    }else{
81      iniciar_events(&(queue[i]));
82      //queue[i].counters[1] = queue[i].instruccionesTotales;
83    }
84  }
85  if(planificar==1){
86    errorPred = predecirDSCR();
87    if(errorPred<0){//si hay errores
88      /*
89        MARCAMOS TERMINADO SI HAY ERROR PARA NO PERDER LOS QUANTUMS VIRTUALES
90      */
91        planificar = 0;
92        fprintf(stderr, "Error de escritura de la prediccion %d\n", errorPred);
93    }
94  }
95  return ret;
96 }
```

El código anterior, corresponde a una función del programa planificador (el código completo está disponible en el Anexo A) que nos permite configurar y medir en cada intervalo, los diferentes eventos. Básicamente, tras la ejecución de cada intervalo virtual, esta función es llamada y se actualiza la variable global *virtualCount* (l. 43-47), que se encuentra inicializada a 0. Se finalizan los contadores actuales con la función *finalitzar_evets* (l. 51). En función del valor de la variable global *virtualCount* (l. 54-66), se escogen los eventos para el próximo intervalo. Más tarde se configuran en el sistema los nuevos eventos en los contadores de prestaciones con la función *iniciar_contadors* (l. 70). De la línea 74 a 83, el código simplemente comprueba que las aplicaciones no hayan terminado todas las instrucciones, y si han terminado su ejecución, pero no han llegado al número máxi-

mo de instrucciones a ejecutar, se relanzan. Por último, de la línea 87 a 94, es un código que usaremos más tarde en la implementación dinámica, pero que se ha decidido ejecutar en la obtención de datos para que las condiciones de ejecución sean lo más parecidas posibles, ya que cada línea de código que ejecuta un procesador puede influir luego en el uso de memoria.

Posteriormente, cuando ya se poseen los datos y el modelo está entrenado con éstos (en la próxima sección se explica como), se puede realizar un análisis de los eventos más a fondo. Para ello se ha empleado una herramienta de la plataforma *BigML* que permite analizar la relación entre dos *inputs*. Por ejemplo la Figura 5.2 muestra la relación entre la configuración de prebúsqueda en el eje X (valor de DSCR) y el evento *PM_L1_ICACHE_RELOADED_PREF* (eje Y). La figura presenta distintos columnas para cada una de las configuraciones de prebúsqueda. Valores distintos se representan con colores distintos, por ejemplo el valor verde representa valores altos y el violeta valores más pequeños. Desde el punto de vista del modelo, que los valores de las configuraciones sean distintos para un determinado parámetro significa que dicho parámetro es útil como característica que permite diferenciar configuraciones distintas y, por tanto, puede ser una entrada adecuada para el modelo. Por el contrario, el mismo color o similar en dos columnas significa que ambas configuraciones toman valores cercanos. En la figura 5.2 se aprecia que algunas configuraciones toman valores distintos; por ejemplo, las configuraciones *OFF* (1 en la figura) y *U7P2* (450), y otras valores similares como la *U7P2* (450) y *U7P7* (455).

Figura 5.2: Espacio que representa la relación entre el evento *PM_L1_ICACHE_RELOADED_PREF* y las distintas configuraciones.

5.2.2. Diseño de la red neuronal artificial

Una vez obtenidos los datos suficientes para implementar una red neuronal artificial, se debe analizar qué datos hay disponibles y cuáles se pueden generar a partir de éstos que nos puedan ser útiles para alimentar la red (*Inputs*), es decir, crear relaciones nuevas con los datos existentes para generar nuevos datos que resulten interesantes para el modelo. También hay que decidir en este punto cuál será la salida (*Output*) de la misma.

En este punto contamos con la información de doce eventos hardware distintos, además del número de instrucciones ejecutadas y ciclos consumidos, sumando un total de catorce eventos. Además, se conoce qué configuración de la prebúsqueda se está empleando en cada ejecución, lo que también puede ser un dato importante a tener en cuenta. Sumando a estos, existe la posibilidad de aplicar la métrica descrita en el capítulo 4 y obtener el rendimiento (IPC) en cada uno de los intervalos. También se puede añadir el rendimiento de intervalos anteriores a estos contadores, es decir el IPC que se obtuvo en el intervalo anterior, para que viendo el rendimiento actual y el anterior, la precisión de la predicción pueda ser más exacta, al poseer más información sobre el histórico. Con todo esto se puede establecer una relación entre los eventos, junto con la configuración empleada para sacar dichos valores, y además cuál es el rendimiento obtenido en el momento de obtener dichos datos y el de intervalos anteriores.

En el presente trabajo, para el desarrollo de la red neuronal artificial se emplea la plataforma *BigML* que proporciona una *API* potente para entrenar redes neuronales y analizar datos de forma sencilla y rápida. Con esta *API* se pueden realizar muchas pruebas en poco tiempo, ya que todo el procesado de entrenamiento requerido para una red neuronal artificial se realiza en la nube, en supercomputadores muy potentes. En nuestro caso, para implementar una red neuronal artificial con la información que tenemos, optaremos por un aprendizaje supervisado, ya que tenemos la posibilidad de etiquetar el rendimiento de nuestros datos, para que la red neuronal artificial aprenda.

Dicho esto, se procede a exponer cuáles son los *inputs* elegidos y cual será el *output* para la propuesta, basada en machine learning, de configuración dinámica de la prebúsqueda realizada en este trabajo:

- ***Inputs***: Se usan los valores de los doce eventos medidos, además de la configuración empleada (DSCR) para tomar dichos valores, y el IPC obtenido en el último intervalo.
- ***Output***: Como se ha indicado, el modelo debe predecir el IPC que obtendrá nuestro sistema para una determinada configuración de prebúsqueda. En otras palabras, la salida de nuestra red será el IPC estimado que alcanzará la aplicación en el próximo intervalo.
- ***Etiquetas***: Además de las entradas mencionadas, para entrenar se utiliza el IPC que obtiene el sistema en ejecución en el siguiente intervalo. Este valor se utiliza como «etiqueta», es decir, como se ha descrito previamente, se utiliza para determinar la relación entre los *inputs* y el valor de la salida, y ayudar en el aprendizaje.

Así pues, una vez obtenidos los datos, los normalizamos para eliminar posible ruido que perjudicaría el entrenamiento. Por ejemplo, se eliminan los datos de aquellos intervalos que ocurren entre transiciones (finalización y relanzamiento de una misma aplicación) cuyos valores son cero. Una vez realizado este paso, procedemos a subir los datos a la plataforma que los convertirá en un «dataset». En nuestro caso también hemos decidido configurar el tipo de datos que corresponden cada uno de los *inputs*. Todos son valores numéricos exceptuando la configuración, DSCR, que es un campo categórico, es

decir, sólo puede tomar un número limitado de valores. En nuestro caso los seis valores corresponden a las seis configuraciones mencionadas. Haciendo esto mejoramos la precisión de la red ya que no debe tener en cuenta los posibles valores que existan entre ellos, ya que para su configuración empleamos valores numéricos. Una vez tenemos todos los datos preparados, desde la plataforma los tenemos que dividir aleatoriamente para test y entrenamiento.

- **Datos de entrenamiento (*training*)**: son empleados para que el algoritmo de aprendizaje obtenga los parámetros del modelo, aplicando las transformaciones explicadas en el capítulo 3. Este conjunto de datos corresponde, en el presente trabajo, con el 80 % de los datos totales obtenidos.

- **Datos de prueba (*test*)**: son los que se guardan para evaluar posteriormente la precisión del modelo con datos nuevos que nunca ha visto, pudiendo medir su tasa de aciertos y fallos a partir del campo «etiqueta», que representaría el valor correcto. Este conjunto de datos corresponde, en este caso, con los restantes datos, es decir del 20 % de los datos totales que se han obtenido.

5.3 Evaluación de la red neuronal artificial

Una vez diseñada y entrenada la red, para comprobar la bondad de una implementación, se usan los datos que se separaron para test, el 20 % de los datos totales, y así probar su rendimiento en las predicciones. En este trabajo aplicaremos tres *funciones de loss* para medir el rendimiento de la red implementada.

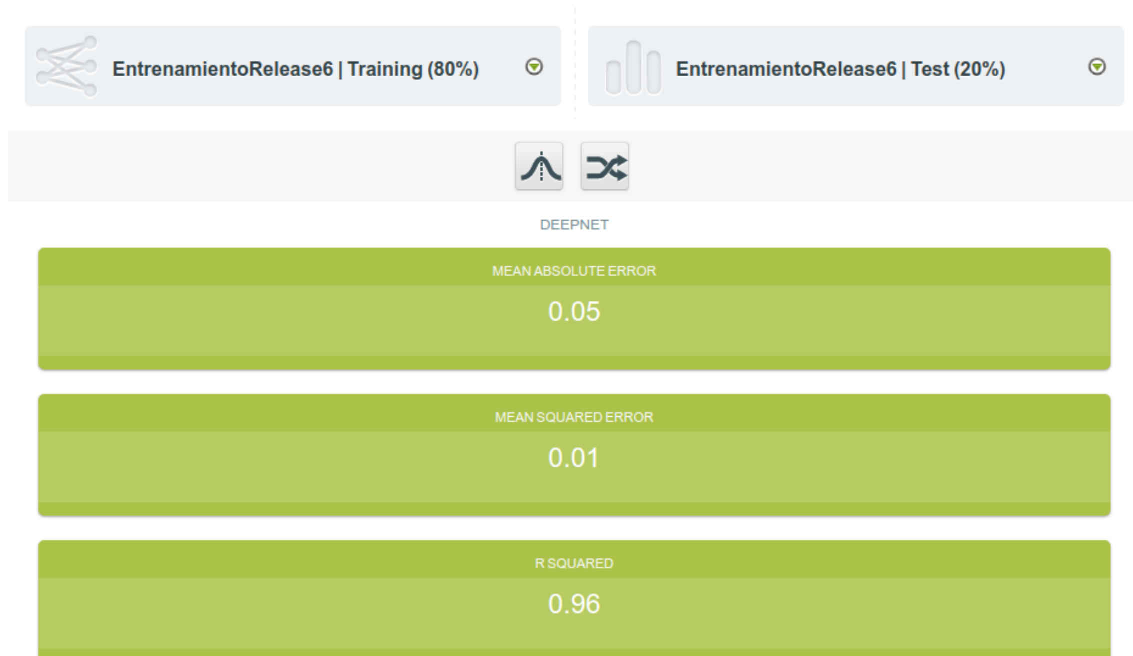


Figura 5.3: Resultados de la evaluación del modelo con los datos de test.

- **Error absoluto medio (MAE)**: esta función realiza una resta entre el valor real con el valor predicho y aplica el valor absoluto para convertirlos en valores positivos. Finalmente, se calcula la media de entre todos los errores absolutos registrados, aplicando la siguiente ecuación:

$$MAE = \frac{\sum_{i=1}^{n} abs(y_i - y_i')}{n}$$

Donde y_i se refiere al valor medido del IPC en el intervalo i (etiqueta) e y_i' se refiere al valor del IPC predicho.

Dado el conjunto de datos de test, el error absoluto promedio de un modelo, se refiere a la media de los valores absolutos de cada error de predicción en todas las instancias del conjunto de datos de prueba. El error de predicción, es la diferencia entre el valor real y el valor predicho para esa instancia. En el caso de nuestra implementación, aplicando esta función obtenemos un error absoluto medio de sólo un 0.05 (5 %) de los datos de test.

- **Error cuadrático medio (MSE)**: Se define mediante la ecuación:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - y_i')^2$$

Esta función básicamente mide el error cuadrado promedio de nuestras predicciones. Para cada punto, calcula la diferencia cuadrada entre las predicciones y el objetivo, y luego promedia esos valores. Cuanto mayor sea este valor, peor es el modelo. Nunca es negativo, ya que estamos elevando al cuadrado los errores de predicción individuales antes de sumarlos, pero sería cero para un modelo perfecto. En esta función la implementación obtiene un 0.01 (1 %) de error.

- **R Squared (R^2)**: determina la calidad del modelo para replicar los resultados, y la proporción de variación de los resultados que puede explicarse por el modelo. El coeficiente de determinación, o R^2, es otra medida que puede usarse para evaluar un modelo y está estrechamente relacionada con la MSE, pero tiene la ventaja de estar libre de escala, no importa si los valores de salida son muy grandes o muy pequeños. El R^2 siempre estará entre $-\infty$ y 1. Cuando R^2 es negativo, significa que el modelo es peor que predecir la media. Formulado sería:

$$R^2 = 1 - \frac{MSE(model)}{MSE(Baseline)}$$

La MSE del modelo se calcula como se ha mostrado ates, mientras que la MSE de la línea de base se define como:

$$MSE(Baseline) = \frac{1}{n}\sum_{i=1}^{n}(y_i - y_i'')^2$$

Donde y''_i es la media de la y observada. Para dejarlo más claro, se puede considerar a esta MSE de referencia como la MSE que obtendría el modelo más simple posible. El modelo más simple posible sería predecir siempre el promedio de todas las muestras. Un valor cercano a 1 indica un modelo con error próximo a cero, y un valor cercano a cero indica un modelo muy próximo a la línea de base. En conclusión, R^2 es la proporción entre lo bueno que es nuestro modelo y lo bueno que es el modelo medio ingenuo. En el caso de la red que se ha implementados se obtiene un valor de 0.96 (96 %), valor muy cercano a 1, con lo que se prueba una vez más que la implementación es bastante fiable.

Como se ha visto en las pruebas de test, la red ha sido capaz de predecir prácticamente el IPC en base a los datos nuevos (no vistos con anterioridad) con los que no se había entrenado. Por tanto, se puede decir que no sufre de sobreajuste (*overfitting*), ya que en dicho caso no habría sido lo suficientemente precisa en estas predicciones ante datos nuevos.

5.4 Implementación de la red neuronal artificial en el planificador

Una vez entrenada la red neuronal artificial y evaluada su precisión, se procede a describir como se ha implementado en el programa planificador escrito en C. Este programa será el encargado de ejecutar una aplicación intervalo a intervalo, monitorizando los eventos necesarios por la red neuronal artificial para predecir la mejor configuración de la prebúsqueda. Esta configuración será aplicada para el siguiente intervalo de la aplicación. El programa seguirá realizando los mismos pasos hasta que la ejecución de la aplicación finalice. El código completo de esta sección se puede encontrar en los Anexos A y B.

```
int predecirDSCR () {
  int i, j;
  DSCRpredict = -1;
  /*
      CALCULAR EL IPC ACTUAL DEL INTERVALO
    */
  actualInstrucciones = instrucciones - actualInstrucciones;
  actualCycles = cycles - actualCycles;

  IPCactual =  (float) actualInstrucciones / actualCycles;

  actualCycles = cycles;
  actualInstrucciones = instrucciones;
  /*
    ESCRIBIR LOS DATOS ACTUALES
  */
  FILE* escribirDatos;
  escribirDatos = fopen("predictionData.csv", "w");
  if (escribirDatos == NULL){
    fprintf(stderr,"Error abriendo predictionData.csv\n");
    return -2;
  }
  fprintf (escribirDatos, "%d,", 0);//El 0 es la primera config
  fprintf (escribirDatos, "%f,", IPCactual);//El 0 es la primera config
  for(j=0; j < numCounters; j++) {
    if(j==numCounters-1){
      fprintf (escribirDatos,"%"PRIu64"", misContadores[j]);
    }else{
      fprintf (escribirDatos, "%"PRIu64",", misContadores[j]);
    }
  }
  fclose(escribirDatos);
  /*
    LLAMADA AL CODIGO DE PREDICCION
  */
  system("cat predictionData.csv | ./redPrediccion.py");
  /*
    LEER LA PREDICCION DSCR
  */
  FILE* leerDSCR;
  leerDSCR = fopen("predictionDSCR.txt", "r");
```

```
42    if (leerDSCR == NULL){
43      fprintf(stderr,"Error abriendo predictionDSCR.txt\n");
44      return -2;
45    }
46    fscanf(leerDSCR, "%d", &DSCRpredict);
47    fclose(leerDSCR);
48    /*
49      LEER LA PREDICCION IPC.
50    */
51    FILE* leerIPC;
52    leerIPC = fopen("predictionIPC.txt", "r");
53    if (leerIPC == NULL){
54      fprintf(stderr,"Error abriendo predictionIPC.txt\n");
55      return -2;
56    }
57    fscanf(leerIPC, "%f", &IPCpredict);
58    fclose(leerIPC);
59    /*
60      MOSTRAMOS LA PREDICCION Y EL ESTADO ACTUAL.
61    */
62    fprintf(stderr, "ACTUALDSCR : DSCR = %d\n", DSCRactual);
63    fprintf(stderr, "ACTUALIPC : IPC = %f\n", IPCactual);
64    fprintf(stderr, "PREDICCIONDSCR : DSCR = %d\n", DSCRpredict);
65    fprintf(stderr, "PREDICCIONIPC : IPC = %f\n", IPCpredict);
66    /*
67      CONFIGURAMOS EL PREFETCHER.
68    */
69    //Comprueba que se haya elegido una config y si no es la que hay actualmente
        configurada, la configura.
70    if(DSCRpredict != -1 && DSCRpredict != DSCRactual){
71      DSCRactual = DSCRpredict;
72      for (i=0; i<N; i++) {
73          //lamar a do dscr pid y comprovar a que no sea -1
74          if(planificar == 1 && queue[i].pid != -1){
75            if(DSCRpredict != -1){
76              if(queue[i].dscr != DSCRpredict){
77                queue[i].dscr = DSCRpredict;
78                do_dscr_pid(queue[i].dscr, queue[i].pid);
79              }
80            }
81          }
82      }
83    }
84    /*
85      MARCAMOS COMO QUE HEMOS TERMINADO DE CONFIGURAR
86    */
87    planificar = 0;
88
89    return 0;
90  }
```

Tal como se presentaba en el primer código mostrado en este capítulo, cuando se recogen los datos necesarios para realizar una predicción, pasados todos los intervalos virtuales, se hace una llamada a la función *predecirDSCR*, presentada en el código de arriba. Esta función lo que hace, en pocas palabras, es preparar los datos y el formato necesarios para llamar a la red, recoger las predicciones, y en última instancia, cambiar la configuración del DSCR si se predice una distinta a la configurada actualmente.

Las líneas de la 7 a la 13, lo que se hace es obtener los ciclos y las instrucciones del último intervalo y aplicando la métrica que hemos establecido para medir el rendimiento, se calcula el IPC, que es uno de los *inputs* de la red, el IPC del intervalo anterior. Luego, de la línea 17 a la 32 se escriben en un fichero los datos necesarios en el orden y formato

en el que nuestro programa *Python* los usará para llamar al modelo. La línea 36 es la encargada de ejecutar el programa *Python* que incluye el código necesario para ejecutar la red, pasándole el fichero con los datos. Las siguientes líneas de la 40 a la 58, se encargan de obtener los resultados de la predicción que devuelve la red, el IPC que se prevé y la mejor configuración. De la línea 70 a 83 se configura el *prefetch* si la configuración que devuelve nuestro modelo es distinta a la establecida en ese momento. Por último, la línea 87 marca la variable global *planificar* a 0 indicando que ha terminado la planificación, y no se volverá ha llamar a esta función hasta que esta variable cambie de valor.

Como se ha comentado en la sección anterior, el modelo del presente trabajo predice el IPC (rendimiento) que tiene el sistema según la configuración DSCR empleada. Por tanto, el programa *Python* que realiza la llamada a la red, devolverá aquella configuración en la que la predicción del IPC sea mayor, para posteriormente aplicarla en el caso de que sea distinta a la actual. Por consiguiente, se realizan tantas llamadas a la red como configuraciones se emplearon en el entrenamiento y la que mayor rendimiento se prevea que alcance, esa será la que se configure. De tal forma que el código de la llamada al modelo escrito en lenguaje *Python* quedará tal que así (Código completo en el anexo B):

```
1  csv = CSVInput();
2  DSCRs = [0,1,66,71,450,455];
3  for datos in csv:
4   prediccionMax = -1;
5   dscrMax = -1;
6   for dscr in DSCRs:
7    datos['DSCR'] = dscr;
8    prediccion =
9    deepnet.predict(datos);
10   if (prediccion > prediccionMax):
11    dscrMax = dscr;
12    prediccionMax = prediccion;
```

En el código existe un *array* de configuraciones, *DSCRs*, que contiene los valores numéricos de las posibles configuraciones del *prefetch*. Se recorre este *array* para preguntar a la red cuál sería el rendimiento del sistema con los datos actuales, almacenados en la variable *datos*, y cada una de las configuraciones de este *array*. Se almacena en las variables *dscrMax* la mejor configuración y *prediccionMax* el valor aproximado del IPC para dicha configuración (l. 10, 11 y 12), este último valor servirá de ayuda más tarde para ver cuanto se desvían las predicciones de la realidad, y saber que no son valores aleatorios que por casualidad puedan beneficiar el rendimiento del *prefetch*.

CAPÍTULO 6
Resultados experimentales

6.1 Análisis de los resultados

Aunque el modelo implementado en el capítulo anterior en las *funciones de loss* aplicadas al mismo, presente buenos resultados, la única forma de saber con certeza si es buena dicha implementación, es probándola en un entorno real.

En esta sección se estudian los resultados de la implementación ejecutándose en la máquina real descrita en el capítulo 4. En los gráficos que se observan a continuación, tenemos el IPC alcanzado por cada una de las configuraciones vistas anteriormente (Tabla 4.2) siendo la configuración denominada *auto* la de nuestra implementación. Se analizarán los dos casos que pueden ocurrir, en el comportamientos de las aplicaciones frente a las configuraciones, que se vieron en el capítulo anterior.

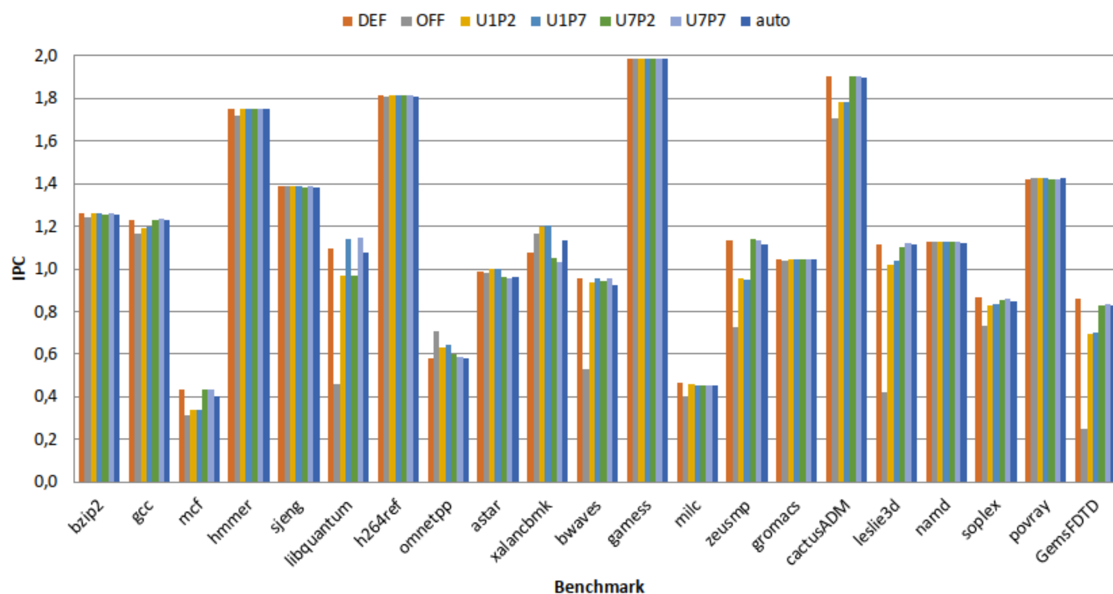
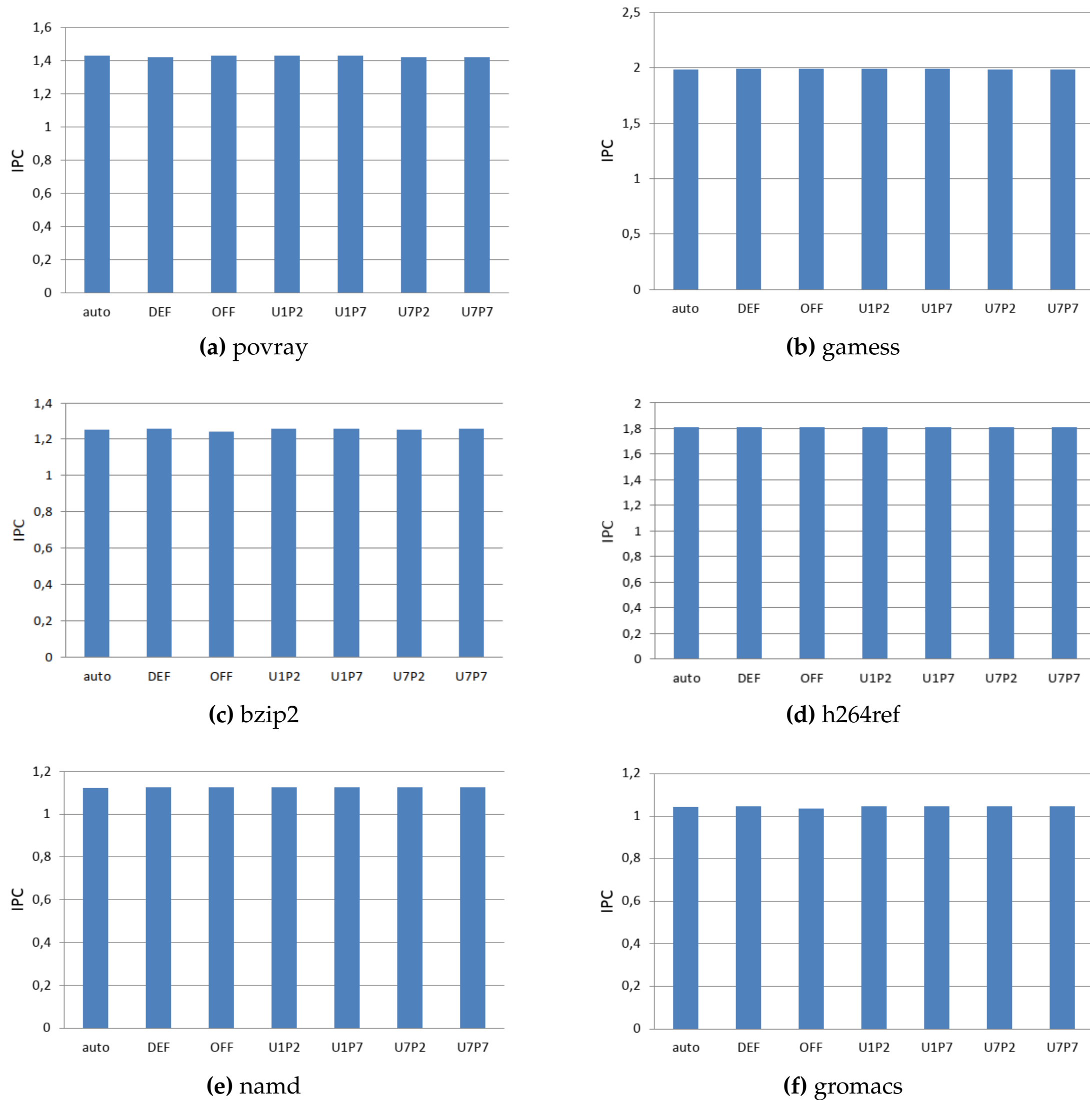


Figura 6.1: Prestaciones de las configuraciones estudiadas y nuestra implementación (*auto*) para los distintos *benchmarks*.

Aplicaciones insensibles a la configuración de *prefetch*

Por una parte, como se ha mencionado en capítulos anteriores, tenemos varias aplicaciones dónde la configuración no aportaba beneficios en el rendimiento, como eran *gamess*, *h264ref* o *povray*. Sabiendo esto se deduce que no se pueden mejorar sus presta-

ciones, usando cualquiera de las configuraciones vamos a obtener los mismos resultados. Un ejemplo de este comportamiento lo presenta la aplicación *povray*, tal y como se muestra en la figura 6.2a. Consecuentemente lo mismo ocurre con las aplicaciones *gamess* y *h264ref*.



(a) povray **(b)** gamess

(c) bzip2 **(d)** h264ref

(e) namd **(f)** gromacs

Figura 6.2: Rendimiento de las aplicaciones insensibles a la configuración del *prefetch*.

Aunque estas aplicaciones no sean sensibles al *prefetch*, para este estudio los resultados que se observan en las figuras 6.3 sirven como un primer indicativo de que la red esta prediciendo el rendimiento correctamente. En ellas, se compara como evoluciona el rendimiento (IPC) predicho por la red (línea azul) respecto al rendimiento real obtenido (línea naranja). Se puede ver como los valores predichos en comparación al valor real siguen una tendencia similar, indicándonos que la red es capaz de identificar la tendencia en la que evoluciona el IPC, aunque en algunos casos se distancian los valores reales y predichos. Por ejemplo, la aplicación *bzip2* (Figura 6.3a) consigue una superposición de las líneas casi perfecta. Sin embargo, *gamess* (Figura 6.3b), consigue obtener la tendencia, pero predice un IPC algo inferior. El tercer caso expuesto (Figura 6.3c) muestra otra vez un seguimiento de la tendencia, pero con predicciones del IPC algo inferiores al rendimiento real cuando el IPC supera el valor de 1,1.

(a) bzip2

(b) gamess

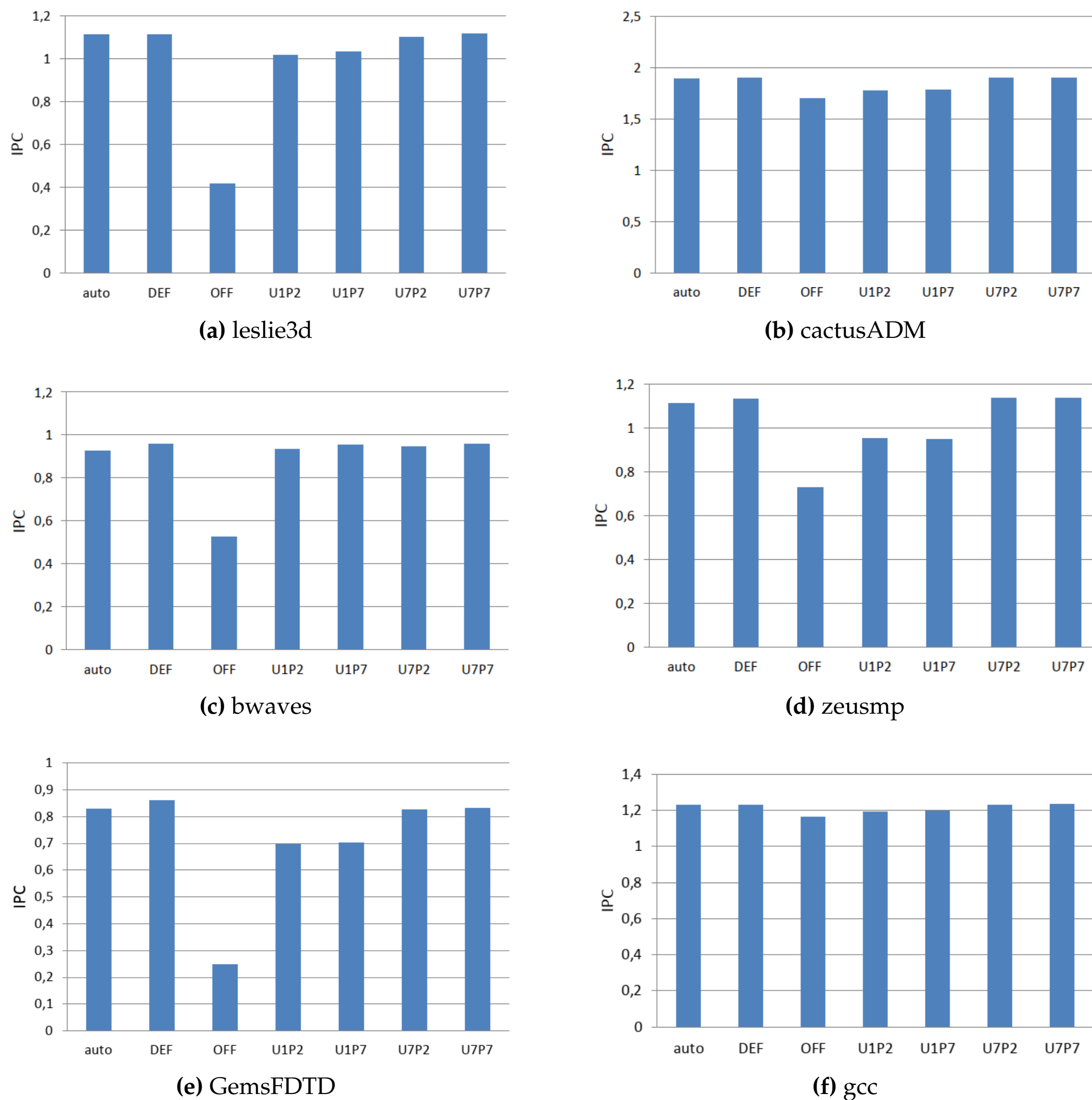
(c) namd

Figura 6.3: Historial de predicciones con respecto al rendimiento real de las aplicaciones insensibles a la configuración del *prefetch*.

Aplicaciones sensibles a la configuración de *prefetch*

Por otro lado, existen aplicaciones en las que la prestaciones varían de manera significativa según la configuración usada. Entre ellas, en capítulos anteriores se identificaron *GemsFDTD*, *leslie3d*, *cactusADM* o *bwaves*.

A continuación se estudian los resultados de las distintas aplicaciónes dependiendo de la sensibilidad de sus prestaciones a la configuración de la prebúsqueda (Figuras 6.4).



(a) leslie3d **(b)** cactusADM

(c) bwaves **(d)** zeusmp

(e) GemsFDTD **(f)** gcc

Figura 6.4: Rendimiento de las aplicaciones sensibles a la configuración del *prefetch*.

Empezando por el gráfico 6.4e su puede apreciar que las mejores configuraciones para dicha aplicación son la configuración por defecto, *DEF*, y las que tienen una mayor urgencia, *U7P2* y *U7P7*. Nuestra propuesta de configuración dinámica (*auto*) alcanza buenas prestaciones, las cuales se quedan muy cerca de la opción por defecto (*DEF* o *U4P4*), que en este caso es la que mejores prestaciones obtiene. La figura 6.1 visualiza las distintas configuraciones predichas y utilizadas por nuestra propuesta (en porcentaje). Como se observa en la figura 6.5e, se ha predicho todo el tiempo la misma configuración, *U7P7*, y por ello los resultados son equivalentes a las de dicha configuración. Esta configuración dista apenas centésimas de la configuración *DEF* por lo que se puede considerar que nuestra propuesta realiza una buena predicción.

(a) leslie3d **(b)** cactusADM

(c) bwaves **(d)** zeusmp

(e) GemsFDTD **(f)** gcc

Figura 6.5: Porcentaje de predicción de las configuraciones del *prefetch* durante la ejecución de las aplicaciones.

La aplicación *leslie3D* es también muy sensible a la configuración del *prefetch*. En este caso se alcanzan las máximas prestaciones al igual que con las configuraciones *DEF*, *U7P2* y *U7P7* (Figura 6.4a). Estas configuraciones mencionadas ofrecen un rendimiento bastante alejado de la configuración sin *prefetch* y algo por encima de las de baja urgencia. Al igual que ocurría con *GemsFDTD*, en este caso también se ha predicho como mejor configuración *U7P7* (Figura 6.5a), que en este caso si alcanza las máximas prestaciones. Esto podría explicar porque en la aplicación anterior se ha escogido también esta configuración en vez de *DEF*, ya que presentan comportamientos similares. La aplicación *zeusmp*

presenta exactamente el mismo comportamiento que esta (Figura 6.4d), aunque en esta última las predicciones han optado por usar parte del tiempo el *prefetch* por defecto y el *prefetch* apagado (Figura 6.5d).

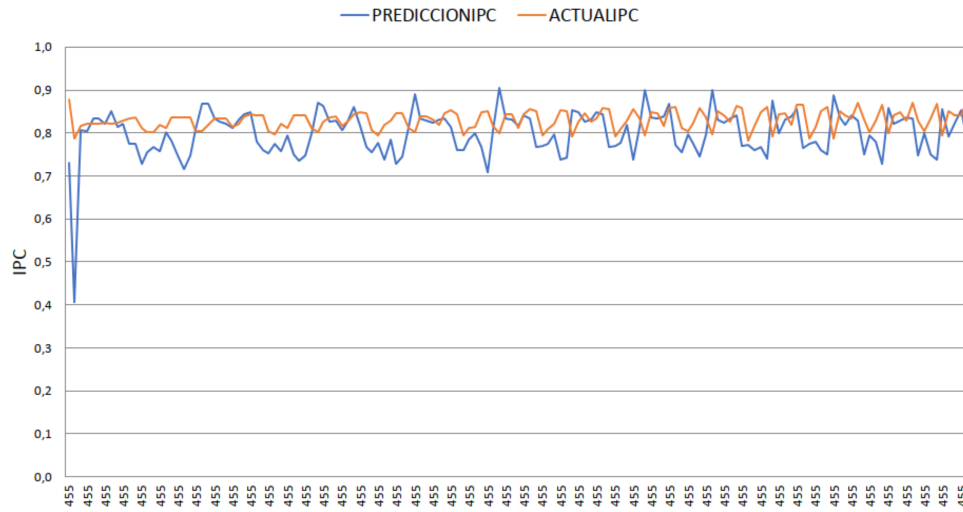
En tercer lugar teníamos la aplicación *cactusADM* (Figura 6.4b), que presentaba prácticamente el mismo comportamiento que los dos casos anteriores, exceptuando que en esta aplicación no existe tanta diferencia entre encender el mecanismo del *prefetch* o dejarlo apagado. Aunque siguen siendo las mejores configuraciones las de urgencia y por defecto. En este caso también conseguimos igualar a la configuración por defecto. Como ocurría en los casos anteriores, al presentar un comportamiento similar, la mejor configuración la mayoría de las veces ha resultado ser *U7P7* (Figura 6.5b), aunque algunas veces ha predicho que la configuración por defecto puede ser válida, cosa que es cierta.

La última de las aplicaciones que conseguían altos beneficios usando técnicas de *prefetch* era *bwaves*. Esta aplicación presenta un comportamiento algo distinto a las tres anteriores, en este caso el uso de cualquiera de las configuraciones de preftch activado saca el mismo rendimiento, aunque estos se distancian bastante de la configuración de preftch desactivado (Figura 6.4c). Sin embargo, cuando vemos el porcentaje de veces que nuestra red ha predicho cada configuración (Figura 6.5c), vemos que la mayoría de las veces ha estado prediciendo que la mejor configuración era *U7P7*, porque en parte compartirá parte del comportamiento de las aplicaciones descritas anteriormente. Aunque también vemos que el 23 % de las veces ha estado prediciendo que la mejor configuración es la de *prefetcher* apagado,pero en las pruebas ha sido con diferencia la que peor rendimiento general ha obtenido. Aún así , nuestra implementación se mantiene a la par con las demás configuraciones en cuanto a rendimiento final se refiere. Lo interesante de estos resultados es que la red ha identificado los momentos en la ejecución de esta aplicación en concreto en los que apagar el *prefetcher* nos aporta mayor rendimiento. Esto puede resultar beneficioso en otros aspectos a pesar de estar de forma casi inapreciable por debajo de la configuración por defecto, ya que al apagar el *prefetcher*, como vimos en el capítulo 3, esta acción conlleva una disminución del uso del ancho de banda del sistema, y en este caso hemos conseguido no usarlo durante el 23 % del tiempo sin perder rendimiento, lo que no se puede hacer sin emplear una planificación dinámica del *prefetcher*.

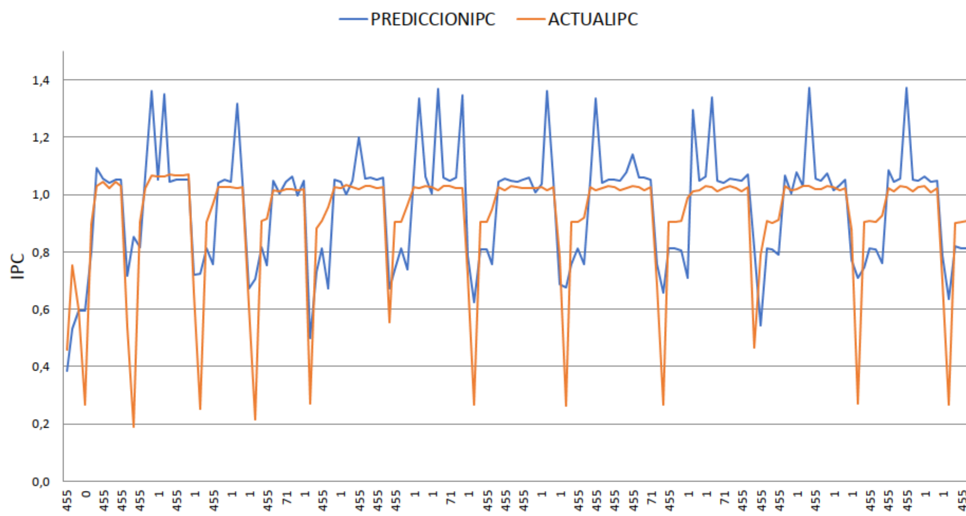
Algo similar a lo que veíamos en *bwaves* ocurre con la aplicación *gcc* (Figura 6.4f), aunque en este caso la mayor parte del tiempo se predice que la mejor configuración es *DEF* (Figura 6.5f).

Un último caso particular de los resultados vistos en la figura 6.1 es que solamente existe una aplicación que se beneficia en gran medida del *prefetcher* apagado, y esta es *omnetpp*. Esta es la aplicación en la que que peores resultados obtenemos y muy probablemente sea fruto de que la red no haya tenido suficientes datos para entrenarse, al solo haber una aplicación en esta suite que presente dicho comportamiento.

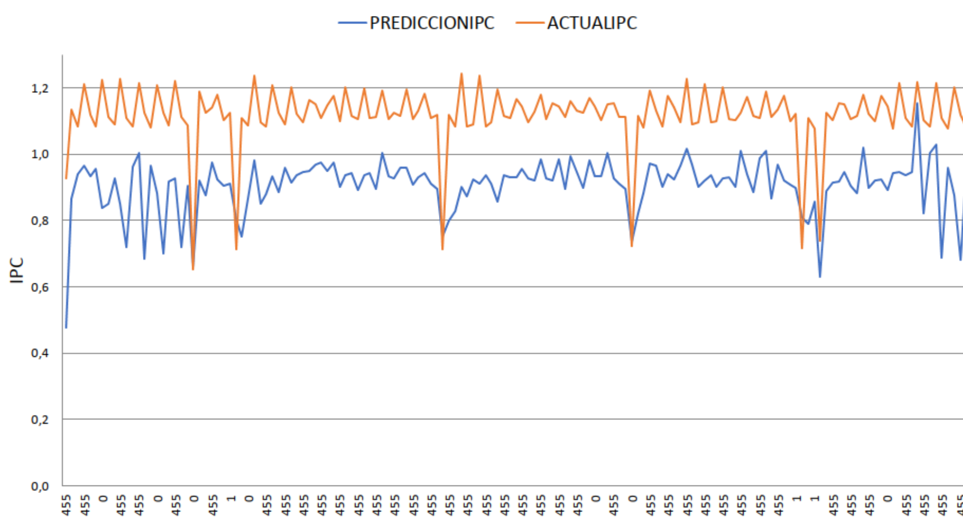
En cuanto a la relación entre las predicciones y los valores reales del IPC. En los tres ejemplos que se presentan en la figura 6.6, se observa una vez más como la red es capaz de identificar la tendencia en la evolución del IPC, prediciendo cuando puede aumentar y cuando disminuir, normalmente con valores muy cercanos a la realidad.



(a) GemsFDTD



(b) bwaves



(c) zeusmp

Figura 6.6: Historial de predicciones con respecto al rendimiento real de las aplicaciones sensibles a la configuración del *prefetch*.

CAPÍTULO 7
Conclusiones

7.1 Visión general del trabajo realizado

Los procesadores modernos implementan diversos *prefetchers* para ocultar la latencia de acceso a memoria, cuya tarea consiste en acercar datos al procesador antes de que éste los requiera. El *IBM POWER8* presenta un *prefetcher* muy avanzado que permite configurar su comportamiento para cada aplicación de manera dinámica, aunque no es una tarea sencilla debido al gran número de combinaciones posibles y a las características de las aplicaciones que influyen en cómo la prebúsqueda afecta a sus prestaciones.

Como hemos observado, podemos obtener algo más de rendimiento con un planificador dinámico que con los de por defecto. Conseguimos que las aplicaciones obtengan un rendimiento similar o superior al de muchas otras configuraciones estáticas, con lo que esperamos obtener mejores resultados en cargas multiprogramadas, beneficiando tanto a las aplicaciones que se benefician de una configuración de *prefetcher* «x», como de otras que se benefician de desactivarlo. Lo más importante de los resultados obtenidos es que vemos que con los datos que nos ofrece el procesador a través de sus eventos o contadores, es viable predecir su rendimiento mediante algoritmos de *machine learning*. Así, profundizado más en la implementación de estos para dicha tarea podemos llegar a conseguir grandes mejoras en la configuración del *prefetcher*.

7.2 Relación con los estudios cursados

Este proyecto ha surgido gracias al interés generado en mí por las arquitecturas hardware al estudiar asignaturas como Estructura de Computadores (ETC), Arquitectura e Ingeniería de Computadores (AIC) y Arquitecturas Avanzadas (AAV). Estas tres áreas despertaron mi curiosidad y ganas de profundizar en el tema, motivándome a buscar información y estudiar con profundidad para obtener los conocimientos necesarios para la comprensión de los temas tratados en el proyecto relacionados con la temática del hardware.

Por otro lado, asignaturas como Sistemas Inteligentes (SIN), Estadística (EST) o Análisis Matemático (AMA), me han permitido entender como funcionan los algoritmos de aprendizaje automático empleados en este proyecto, saber qué pasos hay que seguir para desarrollarlos, y saber analizar sus resultados.

Otro conjunto de asignaturas que me han ayudado en la programación de los códigos para la implementación, los scripts de lanzamiento, el procesado de datos y, sobretodo para entender como configurar el sistema, son Fundamentos de los Sistemas Operativos (FSO), Diseño de Sistemas Operativos (DSO) y Seguridad en los Sistemas Informáticos

(SSI). Durante mi paso por estos cursos he aprendido a programar en *C*, realizar programas usando el *Bash* del sistema *Linux*, y a manejar datos con el lenguaje *Python*.

Participar en un proyecto final de carrera conjuntamente con mis tutores y haber realizado un intercambio académico durante este período, todo esto, ha generado una simulación de lo que podría haber sido un entorno de trabajo profesional. Había que cumplir unos horarios con unas fechas de entrega durante las distintas reuniones que se han realizado. Todo esto que estudié en la asignatura de Gestión de Proyectos (GPR) y también en Ingeniería de Software (ISW), lo he podido vivir en primera persona.

Durante los estudios cursados he tenido que realizar distintas memorias y trabajos prácticos que me han ayudado mucho a mejorar mi expresión escrita y a organizar toda la información, lo que me ha servido de mucha ayuda en este trabajo.

Hablando de las competencias transversales desarrolladas durante los cursos realizados, puedo destacar aquellas que creo que describen muy bien como ha sido la realización del trabajo.

- **CT_02. Aplicación y pensamiento práctico, CT_09. Pensamiento crítico, CT_11. Aprendizaje permanente**: Como en la mayoría de proyectos en los campos relacionados con la ingeniería, es necesario tener estas competencias desarrolladas para poder aplicar los conocimientos teóricos en algo práctico y además poder reflexionar sobre los resultados obtenidos sobre cómo se podrían mejorar y en qué nos hemos podido equivocar.
- **CT_03. Análisis y resolución de problemas,CT_10. Conocimiento de problemas contemporáneos, CT_04. Innovación, creatividad y emprendimiento**: Para realizar este trabajo se ha partido de un problema existente y muy de moda últimamente, cómo es la prebúsqueda. Se ha analizado el problema, estudiando tanto el problema en sí, como las actuales propuestas, y así empezar con la resolución del mismo con una implementación propia.
- **CT_06. Trabajo en equipo y liderazgo, CT_08. Comunicación efectiva, CT_12. Planificación y gestión del tiempo**: Estas competencias se han visto reflejadas en lo que comentaba anteriormente, el hecho de tener que cumplir con unos plazos preestablecidos, comunicar los avances con los tutores y decidir cómo avanzar a raíz de los mismos.
- **CT_13. Instrumental específico**: Como se ha visto en el presente trabajo se han empleado distintas herramientas específicas para su desarrollo, como el sistema IBM, el operativo basado en *GNU/Linux Ubuntu*, procesadores de datos como *excel* o la *API BigML*, lenguajes de programación, compiladores y librerías, entre otras.

7.3 Trabajos futuros

Este trabajo se enmarca dentro de un proyecto del plan de investigación estatal **"Tecnologías Innovadoras de Procesadores, Aceleradores y Redes para Centros de datos y Computación de Altas Prestaciones (T-PARCCA)"**. En este trabajo nos hemos centrado en la configuración dinámica de la prebúsqueda hardware en ejecuciones individuales para estudiar la viabilidad de implementar un planificador de la configuración del *prefetch*, que sea capaz de identificar para cada tipo de carga y cual es su mejor configuración en tiempo real.

La idea es que el presente trabajo sirva como base para la aplicación de técnicas de *deep learning* en sistemas más complejos. En este sentido se plantean múltiples objetivos.

En primer lugar continuar con el estudio de parámetros (entradas y etiquetas) y otros tipos de modelos como redes neuronales recurrentes (RNN) para mejorar la precisión del predictor.

En segundo lugar, nos planteamos desarrollar una versión del planificador para cargas multiprograma compuestas por múltiples aplicaciones individuales ejecutándose en núcleos distintos. En estos sistemas debe considerarse el ancho de banda del sistema de memoria que se convierte en un recurso crítico.

Finalmente, también se prevé realizar el estudio considerando *benchmarks* de ámbitos diferentes, por ejemplo cargas de tipo *cloud*, o en máquinas con procesadores distintos al empleado en este trabajo.

Bibliografía

[1] Dr. Jordi Torres. *DEEP LEARNING Introducción práctica con Keras*. Espasa Calpe, S.A., Madrid, sisena edició, 2008.

[2] Francois Chollet. *Deep Learning with Python*. Espasa Calpe, S.A., Madrid, sisena edició, 2008.

[3] Marti Torrents Lapuerta. *Improving prefetching mechanisms for tiled cmp platforms.* 2016.

[4] Brian Hall, Peter Bergner, Alon Shalev Housfater, Madhusudanan Kandasamy, Tulio Magno, Alex Mericas, Steve Munroe, Mauricio Oliveira, Bill Schmidt, Will Schmidt, et al. *Performance optimization and tuning techniques for IBM Power Systems processors including IBM POWER8.* IBM Redbooks, 2017.

[5] John L Henning. *Spec cpu2006 benchmark descriptions.* ACM SIGARCH Computer Architecture News, 34(4):1–17, 2006.

[6] Vicent Selfa, Julio Sahuquillo, María E. Gómez, Crispín Gómez *Efficient selective multicore prefetching under limited memory bandwidth* Journal of Parallel and Distributed Computing https://doi.org/10.1016/j.jpdc.2018.05.002.

[7] Biswabandan Panda, Shankar Balachandran *Expert Prefetch Prediction: An Expert Predicting the Usefulness of Hardware Prefetchers* http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.705.1406&rep=rep1&type=pdf.

[8] Carlos Navarro, Josue Feliu, Salvador Petit, Maria E. Gómez, Julio Sahuquillo *Mejora de las prestaciones del prefetcher para cargas multiprograma en el IBM POWER8* https://aplicat.upv.es/exploraupv/ficha-publicacion/publicacion/369158.

[9] Shih-wei Liao, Tzu-Han Hung, Donald Nguyen, Chinyen Chou, Chiaheng Tu, and Hucheng Zhou *Machine Learning-Based Prefetch Optimization for Data Center Applications* https://ieeexplore.ieee.org/document/6375514.

[10] Zhen Jia ; Chao Xue ; Guancheng Chen ; Jianfeng Zhan ; Lixin Zhang ; Yonghua Lin ; Peter Hofstee *Auto-tuning Spark big data workloads on POWER8: Prediction-based dynamic SMT threading* https://ieeexplore.ieee.org/document/7756772

[11] Saami Rahman, Martin Burtscher, Ziliang Zong, Apan Qasem *Maximizing Hardware Prefetch Effectiveness with Machine Learning* https://ieeexplore.ieee.org/document/7336192/authors#authors.

[12] Diana Guttman, Mahmut Taylan Kandemir, Meena Arunachalam, Rahul Khanna *Machine learning techniques for improved data prefetching* https://ieeexplore.ieee.org/document/7352208.

[13] Milad Hashemi, Kevin Swersky, Jamie A. Smith, Grant Ayers, Heiner Litz, Jichuan Chang, Christos Kozyrakis, Parthasarathy Ranganathan *Learning Memory Access Patterns* https://arxiv.org/pdf/1803.02329.pdf.

[14] Leeor Peled ; Shie Mannor ; Uri Weiser ; Yoav Etsion *Semantic locality and context-based prefetching using reinforcement learning* https://ieeexplore.ieee.org/document/7284073.

[15] Minghua Li, Guancheng Chen, Qijun Wang, Yonghua Lin, Peter Hofstee, Per Stenstrom, Dian Zhou *PATer: A Hardware Prefetching Automatic Tuner on IBM POWER8 Processor* https://ieeexplore.ieee.org/document/7120125.

APÉNDICE A

Código ejecución de los Benchmarks en lenguaje *C*

```c
/*******************************************************
 **                    Includes                        **
 *******************************************************/

#include <wait.h>
#include <sys/types.h>
#include <inttypes.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <stdarg.h>
#include <errno.h>
#include <unistd.h>
#include <string.h>
#include <stdarg.h>
#include <sys/wait.h>
#include <sys/ptrace.h>
#include <err.h>
#include <sys/poll.h>
#include <sched.h>

#include "perf_util.h"

/*******************************************************
 **                    Defines                         **
 *******************************************************/

#define _GNU_SOURCE
#define N_MAX 10
#define PTRACE_DSCR 44

/*Los 6 tipos de configuraciones que vamos a usar*/
#define DEF 0
#define OFF 1
#define U1P2 66
#define U1P7 71
#define U7P2 450
#define U7P7 455

/*******************************************************
 **                    Structs                         **
 *******************************************************/

typedef struct {
  char *events;
```

```
46    int delay;  // Duracion del quantum (ms)
47    int pinned;
48    int group;
49    int verbose;  // Imprimir datos finales
50    int verbose_q;  // Imprimir datos por quantum
51  } options_t;
52
53  typedef struct {
54    pid_t pid;  // ID del proceso
55    int benchmark;  // Aplicacion a ejecutar
56    int finished; // Han finalizado las instrucciones?
57    int status;
58    int core; // Nucleo en el que se ejecuta
59    int dscr; // Valor de configuracion del prefetcher
60    int current_counter;
61    uint64_t counters [45];
62    uint64_t instruccionesTotales;
63    perf_event_desc_t *fds;
64    int num_fds;
65    cpu_set_t mask;
66  } node;
67
68  /*********************************************************
69   **                    Variables Globales                **
70   *********************************************************/
71
72  node queue [N_MAX]; // Cola donde se almacenan todas las aplicaciones en
        ejecucion
73  static options_t options;
74
75  int end_experiment = 0;
76  int N;  // Numero de aplicaciones en ejecucion
77  int workload; // Numero de la carga
78
79  uint64_t pmu_counters[7];
80  char *events[7];
81
82  int def = 0;
83  int off = 0;
84  /*
85   * DEF = 0
86   * OFF = 1
87   * U1P2 = 66
88   * U1P7 = 71
89   * U7P2 = 450
90   * U7P7 = 455
91   * */
92  int DSCRBench [] = {
93    1,1,1,1,1,1,1,U1P7,1,1,1, //0--10
94    OFF,1,1,1,U7P2,1,1,U7P2,1,0, //11--20
95    1,1,1,1,1,1,1}; //21--27
96
97  /*********************************************************
98   **               Contadores por quantum                 **
99   *********************************************************/
100
101 int planificar = 0;
102 int virtualQuantums = 3;//Numero de quantums que componen un quantum virtual
103 int virtualCount = 0;
104 int numCounters = 12;
105 uint64_t misContadores[12];//numCounters
106 uint64_t instrucciones = 0;
107 uint64_t cycles = 0;
108 uint64_t actualInstrucciones = 0;
```

```
109 uint64_t actualCycles = 0;
110 int DSCRactual = 0;
111 int DSCRpredict = -1;
112 float IPCactual = 0;
113 float IPCpredict = 0;
114
115 /*********************************************************
116  **                     Benchmarks Spec2006                      **
117  *********************************************************/
118
119 char *benchmarks[][200] = {
120   // 0 -> perlbench
121   {NULL, NULL, NULL},
122   // 1 -> bzip2
123   {"/home/malursem/working_dir/spec_bin/bzip2.ppc64", "/home/malursem/
        working_dir/CPU2006/401.bzip2/data/all/input/input.combined", "200", NULL
        },
124   // 2 -> gcc
125   {"/home/malursem/working_dir/spec_bin/gcc.ppc64", "/home/malursem/working_dir
        /CPU2006/403.gcc/data/ref/input/scilab.i", "-o", "scilab.s", NULL},
126   // 3 -> mcf
127   {"/home/malursem/working_dir/spec_bin/mcf.ppc64", "/home/malursem/working_dir
        /CPU2006/429.mcf/data/ref/input/inp.in", NULL},
128   // 4 -> gobmk
129   {"/home/malursem/working_dir/spec_bin/gobmk.ppc64", "--quiet", "--mode", "gtp
        ", NULL},
130   // 5 -> hmmer
131   {"/home/malursem/working_dir/spec_bin/hmmer.ppc64", "--fixed", "0", "--mean",
         "500", "--num", "500000", "--sd", "350", "--seed", "0", "/home/malursem/
        working_dir/CPU2006/456.hmmer/data/ref/input/retro.hmm", NULL},
132   // 6 -> sjeng
133   {"/home/malursem/working_dir/spec_bin/sjeng.ppc64", "/home/malursem/
        working_dir/CPU2006/458.sjeng/data/ref/input/ref.txt", NULL},
134   // 7 -> libquantum
135   {"/home/malursem/working_dir/spec_bin/libquantum.ppc64", "1397", "8", NULL},
136   // 8 -> h264ref
137   {"/home/malursem/working_dir/spec_bin/h264ref.ppc64", "-d", "/home/malursem/
        working_dir/CPU2006/464.h264ref/data/ref/input/
        foreman_ref_encoder_baseline.cfg", NULL},
138   // 9 -> omnetpp
139   {"/home/malursem/working_dir/spec_bin/omnetpp.ppc64", "/home/malursem/
        working_dir/CPU2006/471.omnetpp/data/ref/input/omnetpp.ini", NULL},
140   // 10 -> astar
141   {"/home/malursem/working_dir/spec_bin/astar.ppc64", "/home/malursem/
        working_dir/CPU2006/473.astar/data/ref/input/BigLakes2048.cfg", NULL},
142   // 11 -> xalancbmk
143   {"/home/malursem/working_dir/spec_bin/Xalan.ppc64", "-v", "/home/malursem/
        working_dir/CPU2006/483.xalancbmk/data/ref/input/t5.xml", "/home/malursem
        /working_dir/CPU2006/483.xalancbmk/data/ref/input/xalanc.xsl", NULL},
144   // 12 -> bwaves
145   {"/home/malursem/working_dir/spec_bin/bwaves.ppc64", NULL},
146   // 13 -> gamess
147   {"/home/malursem/working_dir/spec_bin/gamess.ppc64", NULL},
148   // 14 -> milc
149   {"/home/malursem/working_dir/spec_bin/milc.ppc64", NULL},
150   // 15 -> zeusmp
151   {"/home/malursem/working_dir/spec_bin/zeusmp.ppc64", NULL},
152   // 16 -> gromacs
153   {"/home/malursem/working_dir/spec_bin/gromacs.ppc64", "-silent", "-deffnm", "
        /home/malursem/working_dir/CPU2006/435.gromacs/data/ref/input/gromacs", "
        -nice", "0", NULL},
154   // 17 -> cactusADM
155   {"/home/malursem/working_dir/spec_bin/cactusADM.ppc64", "/home/malursem/
        working_dir/CPU2006/436.cactusADM/data/ref/input/benchADM.par", NULL},
```

```
156   // 18 -> leslie3d
157   {"/home/malursem/working_dir/spec_bin/leslie3d.ppc64", NULL},
158   // 19 -> namd
159   {"/home/malursem/working_dir/spec_bin/namd.ppc64", "--input", "/home/malursem
          /working_dir/CPU2006/444.namd/data/all/input/namd.input", "--iterations",
          "38", "--output", "namd.out", NULL},
160   // 20 -> microbench
161   {"/home/malursem/working_dir/microbenchArray160MBinf", "100", "0", "1024", "0
          "},
162   // 21 -> soplex
163   {"/home/malursem/working_dir/spec_bin/soplex.ppc64", "-s1", "-e","-m45000", "
          /home/malursem/working_dir/CPU2006/450.soplex/data/ref/input/pds-50.mps",
          NULL},
164   //22 -> povray
165   {"/home/malursem/working_dir/spec_bin/povray.ppc64", "/home/malursem/
          working_dir/CPU2006/453.povray/data/ref/input/SPEC-benchmark-ref.ini",
          NULL},
166   // 23 -> GemsFDTD
167   {"/home/malursem/working_dir/spec_bin/GemsFDTD.ppc64", NULL},
168   // 24 -> lbm
169   {"/home/malursem/working_dir/spec_bin/lbm.ppc64", "300", "reference.dat", "0"
          , "1", "/home/malursem/working_dir/CPU2006/470.lbm/data/ref/input/100
          _100_130_ldc.of", NULL},
170   // 25 -> tonto
171   {"/home/malursem/working_dir/spec_bin/tonto.ppc64", NULL},
172   // 26 -> calculix
173   {"/home/malursem/working_dir/spec_bin/calculix.ppc64", "-i", "/home/malursem/
          working_dir/CPU2006/454.calculix/data/ref/input/hyperviscoplastic", NULL
          },
174   // 27
175   {NULL, NULL, NULL},
176 };
177 
178 /*********************************************************
179  **                  Nombre de los benchmarks           **
180  *********************************************************/
181 
182 char *benchNames [] = {
183   "perlBench","bzip2","gcc","mcf","gobmk","hmmer","sjeng",  // 0--6
184   "libquantum","h264ref","omnetpp","astar","xalancbmk","bwaves",  // 7--12
185   "gamess","milc","zeusmp","gromacs","cactusADM","leslie3d",  // 13--18
186   "namd","microbench","soplex","povray","gemsFDTD","lbm","tonto","calculix" //
          19--27
187 };
188 
189 /*********************************************************
190  **          Instrucciones a ejecutar por benchmark     **
191  *********************************************************/
192 
193 unsigned long int instruccions_totals [] = {
194   0, 558309327207, 5421059240140,
195   186483654001, 0,776504509655,
196   614626187081, 480070400876,802635200538,
197   261219407437,428862715907,470328894765,
198   428418848680, 886931409787,204234912479,
199   504060702499,463926761740,843934894626,
200   492910117299, 498894476043,87430624497,
201   373477511853,628243699177,376876177856,
202   445124040617,0,0,0
203 };
204 
205 /*********************************************************
206  **                  Tamano de la mezcla                **
207  *********************************************************/
```

```
208
209 int nmezclas [] = {
210   1,  // 0  NO MODIFICAR
211   4,  // 1  NO MODIFICAR
212   8,  // 2
213   8,  // 3
214   10, // 4
215   10, // 5
216   4 // 6
217 };
218
219 /**********************************************************
220  **                  Composicion de la mezcla                **
221  **********************************************************/
222
223 int mezclas [][12] = {
224   {-1}, // 0  NO MODIFICAR
225   {-1,20,20,20},  // 1  NO MODIFICAR
226   {17,15,10,14,3,11,21,9},  // 2
227   {17,15,10,11,3,14,23,21}, // 3
228   {17,17,15,23,11,10,10,3,9,9}, // 4
229   {17,11,21,21,14,14,21,23,23,23},  // 5
230   {7, 11, 18, 15} // 6
231 };
232
233 /**********************************************************
234  **                  do_dscr_pid                             **
235  **********************************************************/
236
237 static int do_dscr_pid(int dscr_state, pid_t pid)
238 {
239   int rc;
240
241   rc = ptrace(PTRACE_ATTACH, pid, NULL, NULL);
242   if (rc) {
243     fprintf(stderr, "Could not attach to process %d to %s the "
244     "DSCR value\n%s\n", pid, (dscr_state ? "set" : "get"),
245     strerror(errno));
246     return rc;
247   }
248
249   wait(NULL);
250
251
252   rc = ptrace(PTRACE_POKEUSER, pid, PTRACE_DSCR << 3, dscr_state);
253   if (rc) {
254     fprintf(stderr, "Could not set the DSCR value for pid "
255     "%d\n%s\n", pid, strerror(errno));
256     ptrace(PTRACE_DETACH, pid, NULL, NULL);
257     return rc;
258   }
259
260   ptrace(PTRACE_DETACH, pid, NULL, NULL);
261   return rc;
262 }
263
264 /**********************************************************
265  **                  iniciar_events                          **
266  **********************************************************/
267
268 void iniciar_events(node *node) {
269   int i, ret;
270
271   // Configurar eventos
```

```
272
273   ret = perf_setup_list_events(options.events, &(node->fds), &(node->num_fds));
274   if (ret || (node->num_fds == 0)) {
275     exit (1);
276   }
277
278   node->fds[0].fd = -1;
279
280   for(i=0; i<node->num_fds; i++){
281     node->fds[i].hw.disabled = 0; /* start immediately */
282     /* request timing information necessary for scaling counts */
283     node->fds[i].hw.read_format = PERF_FORMAT_SCALE;
284     node->fds[i].hw.pinned = !i && options.pinned;
285     node->fds[i].fd = perf_event_open(&node->fds[i].hw, node->pid, -1, (options
          .group? node->fds[i].fd : -1), 0);
286     if (node->fds[i].fd == -1) {
287       errx(1, "cannot attach event %s on node with pid %d", node->fds[i].name,
              node->pid);
288     }
289   }
290 }
291
292 /*********************************************************
293  **                  finalitzar_events                  **
294  *********************************************************/
295
296 void finalitzar_events (node *node) {
297
298   int i;
299
300   // Libera los descriptores
301   for(i=0; i < node->num_fds; i++) {
302     close(node->fds[i].fd);
303   }
304
305   // Libera los contadores
306   perf_free_fds(node->fds, node->num_fds);
307   node->fds = NULL;
308 }
309
310 /*********************************************************
311  **                  iniciar_contadors                  **
312  *********************************************************/
313
314 void iniciar_contadors (node *node) {
315   int i;
316
317   node->current_counter = 0;
318
319   for (i=0; i<45; i++) {
320     node->counters[i] = 0;
321   }
322 }
323
324 /*********************************************************
325  **                    get_countsv2                     **
326  *********************************************************/
327
328 static void get_countsv2(node *aux){
329   ssize_t ret;
330   int i,cont;
331   if(options.verbose_q && !aux->finished){//imprimix si no ha completat totes
        les instruccions
332     switch (virtualCount)
```

```
333      {
334        case 0:
335          fprintf(stderr, "cycles,instructions,PM_L1_ICACHE_MISS,
               PERF_COUNT_HW_CACHE_L1D:READ:MISS,PERF_COUNT_HW_CACHE_L1D:WRITE:
               MISS,PERF_COUNT_HW_CACHE_LL:WRITE:MISS\n");
336          fprintf(stderr, "QuantumCounters0:\t");
337          fprintf(stderr, "%s\t", benchNames[aux->benchmark] );
338          cont =0;//0-3
339          break;
340        case 1:
341          fprintf(stderr, "cycles,instructions,PM_DATA_FROM_L3,PM_MEM_PREF,
               PERF_COUNT_HW_CACHE_L1I:PREFETCH:ACCESS,PERF_COUNT_HW_CACHE_L1D:
               PREFETCH:ACCESS\n");
342          fprintf(stderr, "QuantumCounters1:\t");
343          fprintf(stderr, "%s\t", benchNames[aux->benchmark] );
344          cont =4;//4-7
345          break;
346        case 2:
347          fprintf(stderr, "cycles,instructions,PM_MEM_READ,
               PM_L1_ICACHE_RELOADED_PREF,PM_DATA_FROM_MEMORY,PM_BR_MPRED_CMPL\n
               ");
348          fprintf(stderr, "QuantumCounters2:\t");
349          fprintf(stderr, "%s\t", benchNames[aux->benchmark] );
350          cont =8;//8-9
351          break;
352        default:
353          break;
354      }
355  }
356
357  for(i=0; i < aux->num_fds; i++) {
358    uint64_t val;
359
360    ret = read(aux->fds[i].fd, aux->fds[i].values, sizeof(aux->fds[i].values));
361    if (ret < (ssize_t)sizeof(aux->fds[i].values)) {
362      if (ret == -1)
363      err(1, "cannot read values event %s", aux->fds[i].name);
364      else
365      warnx("could not read event%d", i);
366    }
367
368    val = aux->fds[i].values[0] - aux->fds[i].prev_values[0];
369    aux->fds[i].prev_values[0] = aux->fds[i].values[0];
370    aux->counters[i] += val;
371    if(options.verbose_q && !aux->finished){
372      //fprintf(stderr, "%"PRIu64"\t", val );
373      switch (virtualCount)
374      {
375        case 0:
376          fprintf(stderr, "%"PRIu64"\t", val );
377          break;
378        case 1:
379          fprintf(stderr, "%"PRIu64"\t", val );
380          break;
381        case 2:
382          fprintf(stderr, "%"PRIu64"\t", val );
383          break;
384        default:
385          break;
386      }
387      switch (i)
388      {
389        case 0:
390          cycles = cycles+val;
```

```
391           break;
392         case 1:
393           instrucciones = instrucciones+val;
394           break;
395         default:
396           misContadores[cont] = val;
397           cont++;
398           break;
399       }
400     }
401 
402   }
403 
404   if(options.verbose_q && !aux->finished){
405     fprintf(stderr, "\n");
406   }
407 
408 }
409 
410 /*********************************************************
411  **                  llansar_proces                      **
412  *********************************************************/
413 
414 int llansar_proces (node *node) {
415   FILE *fitxer;
416   pid_t pid;
417 
418   pid = fork();
419   switch (pid) {
420 
421   case -1: //Error
422   //fprintf(stderr, "No he podido crear el hijo.\n");
423   exit(-3);
424 
425   case 0: // Hijo
426 
427   // Descriptores para los que tienen la entrada por la entrada estandar.
428   switch(node->benchmark) {
429 
430     case 4:
431       close(0);
432       fitxer = fopen("/home/malursem/working_dir/CPU2006/445.gobmk/data/ref/
           input/13x13.tst", "r");
433       if (fitxer == NULL) {
434         printf("Error. No se ha podido abrir el fichero arb.tst.\n");
435         return -1;
436       }
437       break;
438 
439     case 13:
440       close(0);
441       fitxer = fopen("/home/malursem/working_dir/CPU2006/416.gamess/data/ref/
           input/h2ocu2+.gradient.config", "r");
442       if (fitxer == NULL) {
443         printf("Error. No se ha podido abrir el fichero h2ocu2+.energy.config.\
             n");
444         return -1;
445       }
446       break;
447 
448     case 14:
449       close(0);
450       fitxer = fopen("/home/malursem/working_dir/CPU2006/433.milc/data/ref/
           input/su3imp.in", "r");
```

```
451        if (fitxer == NULL) {
452          printf("Error. No se ha podido abrir el fichero su3imp.in.\n");
453          return -1;
454        }
455        break;
456
457      case 18:
458        close(0);
459        fitxer = fopen("/home/malursem/working_dir/CPU2006/437.leslie3d/data/ref/
               input/leslie3d.in", "r");
460        if (fitxer == NULL) {
461          printf("Error. No se ha podido abrir el fichero leslie3d.in.\n");
462          return -1;
463        }
464        break;
465
466      case 22:
467        close(2);
468        fitxer = fopen("/home/malursem/working_dir/povray.sal", "w");
469        if (fitxer == NULL) {
470          printf("Error. No se ha podido abrir el fichero povray.sal\n");
471          return -1;
472        }
473        break;
474      }
475
476      execv(benchmarks[node->benchmark][0], benchmarks[node->benchmark]);
477      exit (-2);
478
479      default: // Padre
480
481        usleep(100);
482
483        // Pausamos el proceso
484        kill (pid, 19);
485
486        // Se mira que no haya fallado
487        waitpid(pid, &(node->status), WUNTRACED);
488        if (WIFEXITED(node->status)) {
489          //fprintf(stderr, "ERROR: command process %d exited too early with
                 status %d\n", pid, WEXITSTATUS(node->status));
490          return -2;
491        }
492
493        // Se asigna el pid
494        node->pid = pid;
495
496        // Asignar el core al proceso
497        if (sched_setaffinity(node->pid, sizeof(node->mask), &node->mask) != 0) {
498          //fprintf(stderr, "Sched_setaffinity error: %d.\n", errno);
499          exit(1);
500        }
501
502        // Poner el valor de configuracion de prefetcher del proceso
503        do_dscr_pid(DSCRBench[node->benchmark],node->pid);
504
505
506        return 1;
507    }
508  }
509
510  /*********************************************************
511  **                 INICIO DE LA PREDICCION              **
512  *********************************************************/
```

```
513
514 int predecirDSCR () {
515
516   int i, j;
517   DSCRpredict = -1;
518   /*
519     CALCULAR EL IPC ACTUAL DEL INTERVALO
520     */
521   actualInstrucciones = instrucciones - actualInstrucciones;
522   actualCycles = cycles - actualCycles;
523
524   IPCactual =  (float) actualInstrucciones / actualCycles;
525
526   actualCycles = cycles;
527   actualInstrucciones = instrucciones;
528   /*
529     ESCRIBIR LOS DATOS ACTUALES
530   */
531   FILE* escribirDatos;
532   escribirDatos = fopen("predictionData.csv", "w");
533   if (escribirDatos == NULL){
534     fprintf(stderr,"Error abriendo predictionData.csv\n");
535     return -2;
536   }
537   fprintf (escribirDatos, "%d,", 0);//El 0 es la primera config
538   fprintf (escribirDatos, "%f,", IPCactual);//El 0 es la primera config
539   for(j=0; j < numCounters; j++) {
540     if(j==numCounters-1){
541       fprintf (escribirDatos,"%"PRIu64"", misContadores[j]);
542     }else{
543       fprintf (escribirDatos, "%"PRIu64",", misContadores[j]);
544     }
545   }
546   fclose(escribirDatos);
547
548   /*
549     LLAMADA AL CODIGO DE PREDICCION
550   */
551   system("cat predictionData.csv | ./redPrediccion.py");
552
553   /*
554     LEER LA PREDICCION DSCR
555   */
556   FILE* leerDSCR;
557   leerDSCR = fopen("predictionDSCR.txt", "r");
558   if (leerDSCR == NULL){
559     fprintf(stderr,"Error abriendo predictionDSCR.txt\n");
560     return -2;
561   }
562   fscanf(leerDSCR, "%d", &DSCRpredict);
563   fclose(leerDSCR);
564
565   /*
566     LEER LA PREDICCION IPC.
567   */
568   FILE* leerIPC;
569   leerIPC = fopen("predictionIPC.txt", "r");
570   if (leerIPC == NULL){
571     fprintf(stderr,"Error abriendo predictionIPC.txt\n");
572     return -2;
573   }
574   fscanf(leerIPC, "%f", &IPCpredict);
575   fclose(leerIPC);
576
```

```
  /*
    MOSTRAMOS LA PREDICCION Y EL ESTADO ACTUAL.
  */
  fprintf(stderr, "ACTUALDSCR : DSCR = %d\n", DSCRactual);
  fprintf(stderr, "ACTUALIPC : IPC = %f\n", IPCactual);
  fprintf(stderr, "PREDICCIONDSCR : DSCR = %d\n", DSCRpredict);
  fprintf(stderr, "PREDICCIONIPC : IPC = %f\n", IPCpredict);
  /*
    CONFIGURAMOS EL PREFETCHER.
  */
  //Comprueba que se haya elegido una config y si no es la que hay actualmente
      configurada, la configura.
  if(DSCRpredict != -1 && DSCRpredict != DSCRactual){
    DSCRactual = DSCRpredict;
    for (i=0; i<N; i++) {
        //lamar a do dscr pid y comprovar a que no sea -1
        if(planificar == 1 && queue[i].pid != -1){
          if(DSCRpredict != -1){
            if(queue[i].dscr != DSCRpredict){
              queue[i].dscr = DSCRpredict;
              do_dscr_pid(queue[i].dscr, queue[i].pid);
            }
          }
        }
    }
  }

  /*
    MARCAMOS COMO QUE HEMOS TERMINADO DE CONFIGURAR
  */
  planificar = 0;

  return 0;
}

/*******************************************************
 **                  measure                         **
 *******************************************************/

int measure() {
  int i, ret, errorPred = 1;

  // Libera los procesos
  for (i=0; i<N; i++) {
    if (queue[i].pid > 0) {
      kill(queue[i].pid, 18); //Reanuda Procesos
    }
  }
  // Mira si alguno ha fallado
  for (i=0; i<N; i++) {
    waitpid(queue[i].pid, &(queue[i].status), WCONTINUED);
    if (WIFEXITED(queue[i].status)) {
      //fprintf(stderr, "ERROR: command process %d_%d exited too early with
          status %d\n", queue[i].benchmark, queue[i].pid, WEXITSTATUS(queue[i].
          status));
    }
  }

  // Ejecuta 1 quantum
  usleep(options.delay*1000);


  // Bloquea los procesos
  ret = 0;
```

```
638   for (i=0; i<N; i++) {
639     if (queue[i].pid > 0) {
640       kill(queue[i].pid, 19);
641       //   waitpid(aux->pid, &(aux->status), WUNTRACED);
642     }
643   }
644
645   // Mira si alguno ha fallado
646   for (i=0; i<N; i++) {
647     waitpid(queue[i].pid, &(queue[i].status), WUNTRACED);
648     if (WIFEXITED(queue[i].status)) {
649       //fprintf(stderr, "Process %d_%d finished with status %d\n", queue[i].
            benchmark, queue[i].pid, WEXITSTATUS(queue[i].status));
650       ret++;
651       queue[i].pid = -1;
652     }
653   }
654   // Lectura de contadores
655   for (i=0; i<N; i++) {
656     get_countsv2(&(queue[i]));
657     queue[i].instruccionesTotales += queue[i].counters[1] ; //Sumaho a la
          variable general, si iguales soles son els del quantum
658   }
659   //calculamos el quantum virtual de la siguiente iteracion
660   if(virtualCount < virtualQuantums){
661     virtualCount++;
662   }else{
663     planificar = 1;
664     virtualCount = 0;
665   }
666
667   for (i=0; i<N; i++) {
668     //finalitzar events
669     finalitzar_events(&(queue[i]));
670     //cambiar contadors
671     if(i==0){
672       switch (virtualCount)
673       {
674         case 0:
675           options.events=strdup("cycles,instructions,PM_L1_ICACHE_MISS,
                PERF_COUNT_HW_CACHE_L1D:READ:MISS,PERF_COUNT_HW_CACHE_L1D:WRITE:
                MISS,PERF_COUNT_HW_CACHE_LL:WRITE:MISS");
676           break;
677         case 1:
678           options.events=strdup("cycles,instructions,PM_DATA_FROM_L3,
                PM_MEM_PREF,PERF_COUNT_HW_CACHE_L1I:PREFETCH:ACCESS,
                PERF_COUNT_HW_CACHE_L1D:PREFETCH:ACCESS");
679           break;
680         case 2:
681           options.events=strdup("cycles,instructions,PM_MEM_READ,
                PM_L1_ICACHE_RELOADED_PREF,PM_DATA_FROM_MEMORY,PM_BR_MPRED_CMPL")
                ;
682           break;
683         default:
684           break;
685       }
686     }
687     // Iniciar contadores
688     iniciar_contadors (&(queue[i]));
689     //##### CARLOS ##### INICIAR EVENTOS Y CONTADORES SI PID != -1.
690     //SI PID==-1 SE DEBE RELANZAR SIEMPRE QUE NO SE HAYA PASADO LAS
          INSTRUCCIONES ( llansar_proces(&queue[i]) )
691     //y luego iniciar contadores y eventos
692     if(queue[i].pid == -1){
```

```
693        if(queue[i].instruccionesTotales < instruccions_totals[queue[i].benchmark
              ] && !queue[i].finished){
694          llansar_proces(&queue[i]);
695          iniciar_events(&(queue[i]));
696        }else{
697          end_experiment++;
698          queue[i].finished = 1;
699        }
700      }else{
701        iniciar_events(&(queue[i]));
702        //queue[i].counters[1] = queue[i].instruccionesTotales;
703      }
704    }
705
706    if(planificar==1){
707      errorPred = predecirDSCR();
708      if(errorPred<0){//si hay errores
709        /*
710          MARCAMOS TERMINADO SI HAY ERROR PARA NO PERDER LOS QUANTUMS VIRTUALES
711        */
712          planificar = 0;
713          fprintf(stderr, "Error de escritura de la prediccion %d\n", errorPred);
714      }
715    }
716
717    return ret;
718  }
719
720  /*********************************************************
721   **                   printFinalValues                    **
722   *********************************************************/
723
724  void printFinalValues(node *aux){
725
726    int i;
727    // Identificador de tipo de contador
728    fprintf(stderr, "FinalCounters:\t");
729    // Nombre del benchmark
730    fprintf(stderr, "%s\t", benchNames[aux->benchmark] );
731    // Todos los contadores
732    for(i = 0; i<aux->num_fds;i++){
733      fprintf(stderr, "%"PRIu64"\t", aux->counters[i] );
734    }
735    // Salto de linea
736    fprintf(stderr, "\n");
737
738  }
739
740  /*********************************************************
741   **                   Usage                               **
742   *********************************************************/
743
744  static void usage(void) {
745    printf("\nUso: 20181106_ExperimentosConCargas \n\n");
746    printf("[-e evento1,evento2,...]\n");
747    printf("-d duracionQuantum (ms)\n");
748    printf("[-v [verbose Final, imprimir valores de los contadores al finalizar
           las instrucciones] ]\n");
749    printf("[-q [verbose Quantum, imprimir valores de los contadores por quantum]
            ]\n");
750    printf("[-h [ayuda]]\n");
751    printf("-A carga [0-> Individual, 1-> Individual junto 3 instancias del
           microbenchmark]\n");
752    printf("[-B benchmark [Solo si -A es 1 o 0]]\n");
```

```
753   printf("[-C configuracionPrefetcher [Solo si -A es 1 o 0] ]\n");
754   printf("[-O [Todo se ejecuta con prefetch apagado, excepto microbenchmark si
          se usa -U]]\n");
755   printf("[-D [Todo se ejecuta con prefetch por defecto, excepto microbenchmark
          si se usa -U]]\n");
756   printf("[-S Stride [Stride que se usa en el microbenchmark, necesario
          estipular si se usa. Valor recomendado 256]]\n");
757   printf("[-N Nops [Numero de nops que ejecuta el microbenchmark, para regular
          la carga.]]\n");
758   printf("[-U configuracionPrefetcher [Se puede poner un valor concreto al
          microbenchmark independientemente de -O o -D]]\n\n");
759   printf("\t\tValores NOP\t\t\n");
760   printf("Consumo BW\t\t*Numero Nops*\t\t\n");
761   printf("100%%\t\t\t*0*\t\t\n");
762   printf("90%%\t\t\t*100000*\t\t\n");
763   printf("80%%\t\t\t*250000*\t\t\n");
764   printf("70%%\t\t\t*500000*\t\t\n");
765   printf("60%%\t\t\t*1000000*\t\t\n");
766   printf("50%%\t\t\t*1500000*\t\t\n");
767   printf("40%%\t\t\t*2000000*\t\t\n");
768   printf("30%%\t\t\t*3000000*\t\t\n");
769   printf("20%%\t\t\t*6000000*\t\t\n");
770   printf("10%%\t\t\t*10000000*\t\t\n\n");
771 }
772
773 /************************************************************
774  **                        MAIN PROGRAM                    **
775  ************************************************************/
776
777 int main(int argc, char **argv) {
778   int c, i, ret, quantums = 0;
779   int individualBench = -1;
780   int individualDSCR = -1;
781   int microbenchDSCR = -1;
782
783   options.delay = 0;
784   options.verbose_q = 0;
785   options.verbose = 0;
786
787   end_experiment = 0;
788
789   N = -1;
790
791   for (i=0; i<N_MAX; i++) {
792     queue[i].benchmark = -1;
793     queue[i].finished = 0;
794     queue[i].pid = -1;
795     queue[i].core = -1;
796     queue[i].instruccionesTotales = 0; //Inicializar a 0
797   }
798
799   //Seleccionar cores predefinidos para N_MAX
800   queue[0].core = 0;
801   queue[1].core = 8;
802   queue[2].core = 16;
803   queue[3].core = 24;
804   queue[4].core = 32;
805   queue[5].core = 40;
806   queue[6].core = 48;
807   queue[7].core = 56;
808   queue[8].core = 64;
809   queue[9].core = 72;
810
811   while ((c=getopt(argc, argv,"ghPODvqe:d:A:B:S:N:C:")) != -1) {
```

```
812      switch(c) {
813        case 'e':
814          options.events = optarg;
815          break;
816        case 'P':
817          options.pinned = 1;
818          break;
819        case 'g':
820          options.group = 1;
821          break;
822        case 'v':
823          options.verbose = 1;
824          break;
825        case 'q':
826          options.verbose_q = 1;
827          break;
828        case 'd':
829          options.delay = atoi(optarg);
830          break;
831        case 'h':
832          usage();
833          exit(0);
834        case 'A':
835          workload = atoi(optarg);
836          N = nmezclas[workload];
837          break;
838        case 'B':
839          individualBench = atoi(optarg);
840          break;
841        case 'C':
842          individualDSCR = atoi(optarg);
843          DSCRactual   = individualDSCR;
844          break;
845        case 'O':
846          off = 1;
847          break;
848        case 'D':
849          def = 1;
850          break;
851        case 'S':
852          // Microbenchmark Stride
853          benchmarks[20][3] = optarg;
854          break;
855        case 'N':
856          //Nop
857          benchmarks[20][2] = optarg;
858          break;
859        case 'U':
860          // Microbenchmark DSCR
861          microbenchDSCR = atoi(optarg);
862          break;
863        default:
864          errx(1, "unknown error");
865      }
866    }
867
868    if (N < 0) {
869      fprintf(stderr, "Error: numero de procesos no especificado.\n");
870      return -1;
871    }
872
873    if (!options.events) {
```

```
874     options.events = strdup("cycles,instructions,PM_L1_ICACHE_MISS,
            PERF_COUNT_HW_CACHE_L1D:READ:MISS,PERF_COUNT_HW_CACHE_L1D:WRITE:MISS,
            PERF_COUNT_HW_CACHE_LL:WRITE:MISS");
875   }
876
877   if (options.delay < 1) {
878     options.delay = 200;
879   }
880
881   if(def){
882     for (i=0; i<N; i++) {
883       DSCRBench[queue[i].benchmark] = 0;
884     }
885   }else if(off){
886     for (i=0; i<N; i++) {
887       DSCRBench[queue[i].benchmark] = 1;
888     }
889   }
890
891   for (i=0; i<N; i++) {
892     queue[i].benchmark = mezclas[workload][i];
893   }
894
895   // Se mira si se ha puesto la carga 0(Solo una aplicacion) o 1(aplicacion
        junto al microbenchmark)
896   if(workload==0 || workload==1){
897     if(individualBench<0 || individualDSCR<0){
898       fprintf(stderr, "Error: No se ha especificado aplicacion ni configuracion
            del prefetch.\n");
899       return -1;
900     }
901     queue[0].benchmark = individualBench;
902     DSCRBench[queue[0].benchmark] = individualDSCR;
903   }
904
905   // Poner la configuracion del prefetcher para el microbenchmark si se ha
        seleccionado alguna
906   if(microbenchDSCR>=0){
907     DSCRBench[20] = microbenchDSCR;
908   }
909
910   // Mirar si falta algun benchmark por asignar o nucleo
911   for (i=0; i<N; i++) {
912     if (queue[i].benchmark < 0) {
913       fprintf(stderr, "Error: Falta algun proceso por asignar benchmark.\n");
914       return -1;
915     }
916     if (queue[i].core < 0) {
917       fprintf(stderr, "Error: Falta algun core por asignar.\n");
918       return -1;
919     }
920   }
921   // Iniciar contadores
922   for (i=0; i<N; i++) {
923     iniciar_contadors (&(queue[i]));
924   }
925
926   // Asignar nucleos
927   for (i=0; i<N; i++) {
928     CPU_ZERO(&(queue[i].mask));
929     CPU_SET(queue[i].core, &(queue[i].mask));
930   }
931
932
```

```
933
934   // Inicializar libpfm
935   if (pfm_initialize() != PFM_SUCCESS) {
936     errx(1, "libpfm initialization failed\n");
937   }
938
939
940   for(i=0; i<N; i++) {
941     llansar_proces(&(queue[i]));
942     iniciar_events(&(queue[i]));
943   }
944
945   do {
946     // Ejecuta un quantum y recoge los valores de ese quantum
947     ret = measure();
948     quantums++;
949
950     // Si algun proceso ha finalizado
951     if (ret) {
952       // Se mira cual de las aplicaciones ha finalizado
953       for (i=0; i<N; i++) {
954         if (queue[i].pid == -1) {
955           // Se leen los contadores antes de finalizarlos
956           get_countsv2(&(queue[i]));
957           finalitzar_events(&(queue[i]));
958           // Si se han completado las instrucciones que debe ejecutar
959           if(queue[i].instruccionesTotales >= instruccions_totals[queue[i].
                benchmark] && !queue[i].finished){
960             end_experiment++;
961             queue[i].finished = 1;
962
963             if(options.verbose){
964               // Imprimir contadores globales del nodo
965               printFinalValues(&(queue[i]));
966             }
967             //break;  // Queremos relanzar las aplicaciones hasta que todas
                  completen las instrucciones
968           }
969           // Relanzamos las aplicaciones que han finalizado
970           llansar_proces (&(queue[i]));
971           iniciar_events (&(queue[i]));
972         }
973       }
974     }
975
976     for (i=0; i<N; i++) {
977       if(queue[i].instruccionesTotales >= instruccions_totals[queue[i].
            benchmark]){
978         // Si no ha finalizado ninguna vez aun
979         if(!queue[i].finished){
980         end_experiment++;
981         queue[i].finished = 1;
982
983         if(options.verbose){
984           // Imprimir contadores globales del nodo
985           printFinalValues(&(queue[i]));
986         }
987
988         }
989         // Si esta vivo matar porque ha superado las instrucciones del
              experimento
990         if(queue[i].pid != -1){
991           kill(queue[i].pid, 9);
992         }
```

```
993         // Revisar contadores antes de finalizar los eventos
994         get_countsv2(&(queue[i]));
995         finalitzar_events(&(queue[i]));
996         // Volver a lanzar el programa de 0
997         llansar_proces (&(queue[i]));
998         iniciar_events (&(queue[i]));
999
1000
1001       }
1002     }
1003
1004
1005   }while (end_experiment < N);
1006
1007
1008   // Finalizamos cualquier proceso que pueda quedar pendiente
1009   for (i=0; i<N; i++) {
1010     if (queue[i].pid > 0) {
1011       kill(queue[i].pid, 9);
1012       finalitzar_events(&(queue[i]));
1013     }
1014   }
1015
1016   // Liberar recursos de libpfg de forma limpia
1017   pfm_terminate();
1018
1019
1020   // Imprimim els resultats
1021   /*for (c=0; c<N; c++) {
1022     fprintf(stderr, "Counters:\t");
1023     fprintf(stderr, "%s\t", benchNames[queue[c].benchmark] );
1024     for(i = 0; i<queue[c].num_fds;i++){
1025       fprintf(stderr, "%"PRIu64"\t", queue[c].counters[i] );
1026     }
1027
1028     fprintf(stderr, "\n");
1029   }*/
1030
1031   return 0;
1032 }
```

APÉNDICE B

Código de la red neuronal artificial en lenguaje *Python*

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

# Requires BigML Python bindings
#
# Install via: pip install bigml
#
# or clone it:
#   git clone https://github.com/bigmlcom/python.git

import time
import sys
import csv
import locale
locale.setlocale(locale.LC_ALL, 'en_US.UTF-8')

from bigml.deepnet import Deepnet
from bigml.api import BigML
# Downloads and generates a local version of the DEEPNET,
# if it hasn't been downloaded previously.
deepnet = Deepnet('deepnet/5d64ed8d42129f7df400105a',
                  api=BigML("malursem",
                            "b8e4882706ee24426b76ce2c7ff75b1eb460c090",
                            domain="bigml.io"))
class CSVInput(object):
    """Reads and parses csv input from stdin

       Expects a data section (without headers) with the following fields: DSCR
           ,IPCanterior,PM_L1_ICACHE_MISS,PERF_COUNT_HW_CACHE_L1D:READ:MISS,
           PERF_COUNT_HW_CACHE_L1D:WRITE:MISS,PERF_COUNT_HW_CACHE_LL:WRITE:MISS
           ,PM_DATA_FROM_L3,PM_MEM_PREF,PERF_COUNT_HW_CACHE_L1I:PREFETCH:ACCESS
           ,PERF_COUNT_HW_CACHE_L1D:PREFETCH:ACCESS,PM_MEM_READ,
           PM_L1_ICACHE_RELOADED_PREF,PM_DATA_FROM_MEMORY,PM_BR_MPRED_CMPL\n'

       Data is processed to fall into the corresponding input type by applying
       INPUT_TYPES, and per field PREFIXES and SUFFIXES are removed. You can
       also provide strings to be considered as no content markers in
       MISSING_TOKENS.
    """
    def __init__(self, input=sys.stdin):
        """ Opens stdin and defines parsing constants

        """
        try:
            self.reader = csv.reader(input, delimiter=',', quotechar='"')
```

```
41
42            self.INPUT_FIELDS = [    "DSCR",
43                                     "IPCanterior",
44                                     "PM_L1_ICACHE_MISS",
45                                     "PERF_COUNT_HW_CACHE_L1D:READ:MISS",
46                                     "PERF_COUNT_HW_CACHE_L1D:WRITE:MISS",
47                                     "PERF_COUNT_HW_CACHE_LL:WRITE:MISS",
48                                     "PM_DATA_FROM_L3",
49                                     "PM_MEM_PREF",
50                                     "PERF_COUNT_HW_CACHE_L1I:PREFETCH:ACCESS",
51                                     "PERF_COUNT_HW_CACHE_L1D:PREFETCH:ACCESS",
52                                     "PM_MEM_READ",
53                                     "PM_L1_ICACHE_RELOADED_PREF",
54                                     "PM_DATA_FROM_MEMORY",
55                                     "PM_BR_MPRED_CMPL"]
56
57            self.INPUT_TYPES = [lambda x: int(locale.atof(x)),
58                                lambda x: int(locale.atof(x)),
59                                lambda x: int(locale.atof(x)),
60                                lambda x: int(locale.atof(x)),
61                                lambda x: int(locale.atof(x)),
62                                lambda x: int(locale.atof(x)),
63                                lambda x: int(locale.atof(x)),
64                                lambda x: int(locale.atof(x)),
65                                lambda x: int(locale.atof(x)),
66                                lambda x: int(locale.atof(x)),
67                                lambda x: int(locale.atof(x)),
68                                lambda x: int(locale.atof(x)),
69                                lambda x: int(locale.atof(x)),
70                                lambda x: int(locale.atof(x))]
71
72            self.PREFIXES = {}
73
74            self.SUFFIXES = {}
75            self.MISSING_TOKENS = ['?']
76        except Exception, exc:
77            sys.stderr.write("Cannot read csv"
78                             " input. %s\n" % str(exc))
79
80    def __iter__(self):
81        """ Iterator method
82
83        """
84        return self
85
86    def next(self):
87        """ Returns processed data in a list structure
88
89        """
90        def normalize(value):
91            """Transforms to unicode and cleans missing tokens
92            """
93            value = unicode(value.decode('utf-8'))
94            return "" if value in self.MISSING_TOKENS else value
95
96        def cast(function_value):
97            """Type related transformations
98            """
99            function, value = function_value
100           if not len(value):
101               return None
102           if function is None:
103               return value
104           else:
```

```
105                 return function(value)
106 
107         try:
108             values = self.reader.next()
109         except StopIteration:
110             raise StopIteration()
111         if len(values) < len(self.INPUT_FIELDS):
112             sys.stderr.write("Found %s fields when %s were expected.\n" %
113                              (len(values), len(self.INPUT_FIELDS)))
114             raise StopIteration()
115         else:
116             values = values[0:len(self.INPUT_FIELDS)]
117         try:
118             values = map(normalize, values)
119             for key in self.PREFIXES:
120                 prefix_len = len(self.PREFIXES[key])
121                 if values[key][0:prefix_len] == self.PREFIXES[key]:
122                     values[key] = values[key][prefix_len:]
123             for key in self.SUFFIXES:
124                 suffix_len = len(self.SUFFIXES[key])
125                 if values[key][-suffix_len:] == self.SUFFIXES[key]:
126                     values[key] = values[key][0:-suffix_len]
127             function_tuples = zip(self.INPUT_TYPES, values)
128             values = map(cast, function_tuples)
129             data = {}
130             for i in range(len(values)):
131                 data.update({self.INPUT_FIELDS[i]: values[i]})
132             return data
133         except Exception, exc:
134             sys.stderr.write("Error in data transformations. %s\n" % str(exc))
135             return False
136 
137 
138 # To make predictions fill the desired input_data in next line.
139 
140 # input_data = {
141 #       "PERF_COUNT_HW_CACHE_LL:WRITE:MISS": 1,
142 #       "PM_DATA_FROM_L3": 1,
143 #       "PERF_COUNT_HW_CACHE_L1D:READ:MISS": 1,
144 #       "PERF_COUNT_HW_CACHE_L1D:WRITE:MISS": 1,
145 #       "IPCanterior": 1,
146 #       "PM_L1_ICACHE_MISS": 1,
147 #       "DSCR": "1",
148 #       "PM_MEM_PREF": 1,
149 #       "PERF_COUNT_HW_CACHE_L1I:PREFETCH:ACCESS": 1,
150 #       "PM_DATA_FROM_MEMORY": 1,
151 #       "PM_MEM_READ": 1,
152 #       "PM_L1_ICACHE_RELOADED_PREF": 1,
153 #       "PERF_COUNT_HW_CACHE_L1D:PREFETCH:ACCESS": 1
154 # }
155 # deepnet.predict(input_data, full=True)
156 #
157 # input_data: dict for the input values
158 # (e.g. {"petal length": 1, "sepal length": 3})
159 # full: if set to True, the output will be a dictionary that includes all the
160 # available information about the prediction. The attributes vary depending
161 # on the ensemble type. Please check:
162 # https://bigml.readthedocs.io/en/latest/#local-deepnet-predictions
163 
164 csv = CSVInput()
165 escribirpredDSCR = open("predictionDSCR.txt","w");
166 escribirpredIPC = open("predictionIPC.txt","w");
167 DSCRs = [0,1,66,71,450,455];
168 for inputData in csv:
```

```
169     if not isinstance(inputData, bool):
170         predMaxIPC = -1;##Representa el valor del IPC
171         predMaxDSCR = -1;##Representa el valor del DSCR
172         for dscr in DSCRs:
173             inputData['DSCR'] = dscr;##Establecemos el DSCR
174             predIPC = deepnet.predict(inputData, full=False)## Llamamos a la
                    red
175             if(predIPC>predMaxIPC):
176                 predMaxDSCR = dscr;
177                 predMaxIPC = predIPC;
178         escribirpredDSCR.write(str(predMaxDSCR));
179         escribirpredIPC.write(str(predMaxIPC));
180 escribirpredDSCR.close();
181 escribirpredIPC.close();
```