UNIVERSIDAD DE INGENIERÍA Y TECNOLOGÍA

CARRERA DE CIENCIA DE LA COMPUTACIÓN

MACHINE LEARNING EN LINUX KERNEL: IMPLEMENTACIÓN DE UN PREDICTOR DE MIGRACIONES FORZADAS EN SCHEDULERS MULTICORE

TESIS

Para optar el título profesional de Licenciado en Ciencia de la Computación

AUTOR:

Mauricio Jorge Pinto Larrea

ASESOR

Jorge Luis Gonzalez Reaño

Lima - Perú

2024

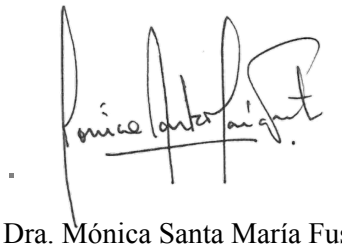
UTEC UNIVERSIDAD DE INGENIERÍA Y TECNOLOGÍA | Dirección de Investigación

DECLARACIÓN JURADA

Yo, Mónica Cecilia Santa María Fuster identificada con DNI No 18226712 en mi condición de autoridad responsable de validar la autenticidad de los trabajos de investigación y tesis de la UNIVERSIDAD DE INGENIERIA Y TECNOLOGIA, DECLARO BAJO JURAMENTO:

Que la tesis denominada "MACHINE LEARNING EN LINUX KERNEL: IMPLEMENTACIÓN DE UN PREDICTOR DE MIGRACIONES FORZADAS EN SCHEDULERS MULTICORE" ha sido elaborada por el señor Mauricio Jorge Pinto Larrea, con la asesoría de Jorge Luis Gonzalez Reaño, identificado con el DNI N°44386274, y que se presenta para obtener el grado de Licenciado en Ciencia de la Computación, ha sido sometida a los mecanismos de control y sanciones anti plagio previstos en la normativa interna de la universidad, encontrándose un porcentaje de similitud de 0%.

En fe de lo cual firmo la presente.



Dra. Mónica Santa María Fuster
Directora de Investigación

En Barranco, el 29 de enero de 2024

Dedicatoria:

Dedico este trabajo a mi madre, quien me ha dado su apoyo incondicional y continúa dándomelo cada día. Gracias por permitirme estudiar y encontrar la carrera que me apasiona, y siempre ayudarme a sentirme seguro de mí mismo. Este trabajo y los demás frutos de mi carrera profesional son y serán gracias a tí.

Además quiero dedicar este trabajo a todas las personas que considero mi familia, sean parientes o amigos. Todos mis logros los comparto con ustedes.

Índice general

Pág.

Índice de tablas

Índice de figuras

RESUMEN

Si bien el *Completely Fair Scheduler* (CFS) de Linux es capaz de proporcionar equidad entre procesos y de manejar la ubicación y migración de estos a través de un *load balancer*, existen trabajos previos que proponen la integración de *Machine Learning* (ML) como una herramienta potencial para refinar las decisiones del *kernel*. En este trabajo, se analiza un caso específico de migración de tareas, donde algunas son migradas forzosamente entre *cores*. Este escenario es subóptimo especialmente cuando, por condiciones específicas, se migra una tarea que se encuentra "caliente" en *cache*.

Como solución a este problema proponemos el uso de ML de manera similar al trabajo [1] con el fin de predecir incidencias de migraciones forzadas. Para esto implementamos un sistema capaz de recolectar datos de migraciones en llamadas a la función `can_migrate_task()` y utilizamos estos datos para (i) entrenar modelos de ML, (ii) realizar inferencias en espacio de *kernel* y (iii) configurar el modelo en tiempo real a través de LibML, una librería que permite hacer uso de redes neuronales de manera híbrida (espacio de usuario y *kernel*).

Los experimentos realizados con modelos entrenados para la predicción de migraciones forzadas muestran que es posible predecir este escenario en espacio de *kernel* con alta precisión alcanzando alrededor del 97 % de exactitud en promedio para todas las cargas de trabajo utilizadas. Adicionalmente, esta implementación no impacta significativamente el *performance* del *kernel*, teniendo el *kernel* modificado un promedio de tiempos de ejecución 2.3 % menor al del original.

Palabras clave:

Linux CFS; Aprendizaje de Máquina; Balance de cargas; Redes neuronales

ABSTRACT

MACHINE LEARNING FOR LINUX KERNEL: STUDY AND IMPLEMENTATION OF A PREDICTOR MODEL OF FORCED MIGRATIONS IN MULTICORE SCHEDULERS USING NEURAL NETWORKS

Although Linux's Completely Fair Scheduler (CFS) is capable of achieving fairness and managing task allocation and migration among cores through a load balancer, recent studies have proposed the use of low level Machine Learning (ML) for optimizing kernel decisions.

In this work, a specific case of scheduling decisions is studied, where tasks are migrated aggressively between cores, either due to being cache-cold, having different NUMA node affinities or having too many failed balance attempts. This is sub-optimal especially when cache-hot tasks are forcefully migrated due to the latter condition being true. In order to solve this problem, this work proposes the use of ML in a way similar to [1] in order to predict incidences of forced migrations. For this, we implemented a system capable of collecting migration related data from calls to the `can_migrate_task()` function and using these to (i) train ML models, (ii) make inferences in kernel space, (iii) configure models in real time through LibML, a library that allows the use of neural networks in kernel and user space in a hybrid manner.

Experiment results where neural networks were trained in userspace with collected migration datasets show that it is possible to predict the occurrence of aggressive migrations with a high precision, reaching accuracy values above 95 % in general terms when running in kernelspace. Additionally, these inferences don't seem to impact performance significantly, as the modified kernel's average runtime for all benchmarks is 2.3 % lower than the original.

Keywords:

Linux CFS; Machine Learning; Load balancing; Neural networks

Capítulo 1

INTRODUCCIÓN

Los sistemas de computación actuales poseen arquitecturas que integran más de un CPU o *core*, comúnmente llamadas arquitecturas *multicore* o *chip multiprocessor* (CMP) [2]. Estos son utilizados en diferentes dominios de la ciencia y la industria, desde los que requieren alta complejidad computacional como *High Performace Computing* (HPC) [3, 4] hasta aplicaciones específicas en sistemas *embedded* y dispositivos móviles [5]. Las arquitecturas *multicore* surgen principalmente durante la década del 2000 debido a dos factores [6]: (1) la necesidad de mantener el crecimiento computacional aumentando el número de transistores de acuerdo a la ley de Moore; y (2) mantener la densidad energética dentro de los límites de la escala de Dennard [7], a pesar del incremento en frecuencia y número de transistores por unidad cuadrada de área en un sólo chip.

El uso de procesadores *multicore* en sistemas de computación motivó también el cambio en los Sistemas Operativos (OS); i.e, sobre la implementación del *scheduler* para la distribución y asignación de tareas entre los *cores* [8–10]. El *scheduler* debe garantizar la equidad o *fairness* entre cargas de trabajo en ejecución, considerando prioridades y permitiendo que todos los procesos progresen en el tiempo de ejecución. Este fue tradicionalmente diseñado para arquitecturas de un sólo procesador (*single core*), de forma que para los años 90 el problema de *scheduling* se consideraba solucionado [8]. Esto cambió con la llegada del procesador *multicore*, añadiendo dos nuevas dimensiones fundamentales al rol del *scheduler*.

Primero, además de multiplexar un core en tiempo compartido (*time sharing*), se volvió necesario lograr un espacio compartido entre procesos (*space sharing*), donde se

debe decidir no solo cuándo sino en qué *core* será asignado un proceso en un intervalo de tiempo [11].

Segundo, una distribución subóptima de tareas entre *cores* genera contención por recursos de bajo nivel compartidos (e.g.,*caches*, controladores y buses), lo cuál impacta significativamente en la *performance* [12].

Actualmente, Linux usa dos técnicas principales de *multicore scheduling* para superar estas limitantes: a) particionamiento, que define colas de ejecución para cada *core* [11, 13, 14], y b) la migración restringida, que limita la cantidad de CPUs a los cuáles una tarea puede migrar [14]. Ambas técnicas se implementan en el *Completely Fair Scheduler* (CFS), usado en *Linux kernel* desde la version 2.6.23 [15]. A pesar de tomar consideraciones sobre el contexto de ejecución y los grupos de CPUs para la distribución de cargas, existe literatura previa [1, 16] que sugiere que las decisiones de CFS para optimizar la distribución de cargas podría refinarse con del uso de técnicas de ML y sus capacidades para predecir escenarios en base a datos históricos como una medida adicional y más precisa que los valores fijos que se utilizan actualmente para representar el estado del sistema, tales como la carga de trabajo y cantidad de procesos en ejecución en cada cola, los estados de cada CPU (activo o inactivo) y el costo de migración de cada tarea [16]. El uso de técnicas de ML en Linux surge como solución a problemas de este tipo, donde la capacidad de predecir el comportamiento de cargas de trabajo del sistema trae consigo beneficios en *performance*. Existen trabajos previos que buscan optimizar las decisiones de componentes del *kernel* como el *scheduler*, sistemas de red y de almacenamiento, implementando técnicas de Machine Learning (ML) y Deep Learning (DL) [1, 16, 17]. Adicionalmente se encuentran trabajos que logran clasificar procesos en base a factores como su tiempo de creación, la memoria que han utilizado, la cantidad de procesadores que utilizan y el tiempo promedio de uso de CPU [18–20], permitiendo hacer predicciones de su comportamiento.

1.1 Descripción de la situación problemática

Las funciones de un sistema operativo (*scheduling*, *memory paging*, *cache mapping*, etc), están dictadas por una gran cantidad de heurísticas fijas en sus programas [21]. Dichas heurísticas no logran adaptarse rápidamente a cambios en la demanda de recursos, ya que esta adaptabilidad requeriría evaluar escenarios previos y patrones de comportamiento para encontrar configuraciones que optimicen sus resultados esperados. Algoritmos de *scheduling* de propósito general como CFS y FreeBSD ULE [22] utilizan estos criterios para definir a qué procesos les corresponde ser ejecutados y en qué *core*, por lo que la integración de ML en esta funcionalidad del kernel para identificar patrones en la distribución y uso de CPUs podría resultar en métodos más precisos y adaptativos.
En el caso de CFS, se propone un criterio para definir si una tarea puede ser migrada de un *core* de origen hacia otro *core* específico de destino bajo ciertas condiciones basadas en parámetros específicos del sistema en aquel instante de tiempo, específicamente cuando a) una tarea tiene mayor afinidad con el nodo de destino, b) existe un exceso de intentos de balance fallidos en un *scheduling domain*, y c) una tarea se encuentra *cache-cold* [23]. Existen casos donde el cumplimiento de algunas de estas condiciones resulta en la migración de una tarea que se encuentra *cache-hot*. Esto es subóptimo ya que en el peor de los casos implicaría una pérdida de afinidad de *cache*, resultando en una migración lenta [24]. Este trabajo propone la predicción de las ocurrencias de esta clase de migraciones a través del uso de ML, y que este podría ser un mecanismo efectivo para evitarlas de antemano, reduciendo pérdidas de *performance* por la migración de una tarea activa en *cache*. La capacidad de aprendizaje que brindan muchas de las técnicas de ML concede una perspectiva adicional a las heurísticas fijas diseñadas para escenarios generales, de manera que el sistema pueda adaptar sus decisiones en base al comportamiento histórico de cargas de trabajo específicas o variables.

1.2 Formulación del problema

Este trabajo se enfoca en el uso de técnicas de ML para la optimización de CFS en base a la predicción del uso de recursos o el ajuste optimizado de ciertos valores. Existen escenarios donde las heurísticas fijas del *kernel* no son lo suficientemente efectivas como para prever decisiones subóptimas, como el *readahead* de páginas de memoria [25] o el enrutamiento de paquetes [26]. En este caso estudiamos un escenario subóptimo: la ocurrencia de migraciones forzadas por un exceso de intentos de balance fallidos; no sólo por el *overhead* de las llamadas fallidas al *load balancer*, sino también por la migración de tareas activas en *cache* que puede llegar a generar un *overhead* adicional por necesidad de migración del entorno de trabajo o *working set migration*. Buscamos responder lo siguiente: si se logran predecir situaciones en las cuáles ocurren migraciones forzadas de tareas activas en *cache*, entonces ¿se pueden reducir las pérdidas que generan migrándolas prematuramente?

1.3 Objetivos de investigación

Son dos los objetivos de este trabajo. Primero, se implementó un modelo para la predicción de las decisiones de balance de cargas de trabajo de CFS tomando en cuenta varios escenarios de ejecución, para así poder migrar prematuramente las tareas que serían migradas agresivamente en futuras llamadas. Esto se divide en cuatro objetivos específicos:

1. Identificar qué variables son las más relevantes para la clasificación de procesos con respecto a su comportamiento y al contexto general de ejecución.

2. Generar un modelo que pueda predecir las ocurrencias de migraciones forzadas, y evaluar la correctitud de estas predicciones a través de un análisis de épocas.

3. Realizar pruebas de precisión y consumo de recursos con cargas de trabajo de diversos *benchmarks*.

Segundo, se evaluó el rendimiento del modelo en *kernel*, probando con diversas configuraciones e implementaciones. Esto se divide en tres objetivos específicos:

1. Implementar el uso de ML en *kernel* de manera que permita predecir las ocurrencias de migraciones agresivas en tiempo real.
2. Medir el *overhead* que genera el uso de una red neuronal para predecir una decisión en la función a optimizar.
3. Con modelos en tiempo real, implementar una librería que logre reajustar modelos en ejecución en *kernel* en base a un entrenamiento en espacio de usuario.

1.4 Justificación

Según nuestro conocimiento, nuestro trabajo es el primero en proponer un modelo de ML como parte del *scheduler* para la predicción de migraciones forzadas. A pesar de aportar con posibles soluciones al complejo problema del *multicore scheduling*, los trabajos previos no estudian la adaptabilidad del sistema frente a escenarios nuevos como cargas de trabajo variables. Este trabajo intenta además expandir la variedad de cargas de trabajo sobre la cuál evalúa sus soluciones y busca hacer evaluaciones no sólo de la precisión del modelo sino también del costo adicional que este genera en el sistema. Existe una extensa cantidad de componentes del *kernel* que pueden ser optimizados haciendo uso de ML, y específicamente del *scheduler*. Parte de los objetivos consiste en realizar pruebas con una variedad de cargas de trabajo que demandan diferentes recursos y cursos de acción del *kernel*. Para probar esto, se toma un enfoque aislado en las migraciones forzadas de tareas entre *cores* por parte del *load balancer* y cómo podrían generar un

overhead adicional. La predicción de estas ocurrencias a través de ML podría proveer un buen indicador para escenarios subóptimos y así forzar las migraciones de manera anticipada evitando pérdidas de rendimiento por llamadas innecesarias a esta rutina.

Capítulo 2

REVISIÓN CRÍTICA DE LA LITERATURA

Existen investigaciones previas que profundizan y proponen soluciones a problemas relacionados con los objetivos de esta investigación, principalmente al uso de ML para mejorar y/o predecir el rendimiento del sistema a través del ajuste de parámetros en puntos focales del *kernel*, y más específicamente a la optimización del balance de cargas del *scheduler*. La mayoría de estos trabajos tienen en común la premisa de utilizar diversas técnicas para optimizar la asignación de recursos ya sea de manera determinística [17] o utilizando modelos probabilísticos o de aprendizaje profundo enfocados en predecir y/o modificar ciertos valores de variables relacionadas al scheduler y los procesos en ejecución [1, 16, 26]. En este capítulo se revisa la literatura relacionada a ambos enfoques.

2.1 Optimizando el scheduler

Empezando por el enfoque determinístico, la metodología del scheduler NEST [17] ataca la falta de consideración de la frecuencia de los cores al momento de asignarles procesos. Los autores señalan cómo CFS se enfoca en el principio de conservación de trabajo, pero sin tomar en cuenta cómo la frecuencia de un core puede afectar también el rendimiento de múltiples procesos y en base a esto proponen NEST, un scheduler que busca maximizar este rendimiento y reducir el consumo energético del sistema en base a dos principios: (i) minimizar la cantidad de cores utilizados de manera que los procesos se mantengan cerca unos de otros y (ii) mantener los cores calientes asegurando actividad prolongada con una alta frecuencia para ser utilizada por los demás procesos.

Durante sus experimentos con NEST, los autores utilizaron la versión 5.9 de Linux en

conjunto con los gobernadores de frecuencia *schedutil* y *performance*, y compararon el rendimiento y consumo energético de NEST contra CFS y otros *schedulers* de vanguardia. Para la mayoría de casos al ejecutar *scripts* de configuración se obtuvo un *speedup* de más del 5 % (a excepción de la congiguración de NodeJS), y siendo el mayor *speedup* de un 37 % en un procesador Intel Xeon E7-8870 v4. Además NEST redujo el consumo energético del CPU entre el inicio y el fin de la ejecución en un 19 %. En general, sus resultados presentan una mejora de rendimiento desde un 10 % hasta un 200 % para aplicaciones con un gran número de *threads* y una reducción de consumo energético en muchos casos.

Dicho trabajo contribuye principalmente brindando un nuevo enfoque orientado a la conservación de dos heurísticas: (i) proximidad entre procesos y (ii) frecuencias altas en los *cores*. Sin embargo, las heurísticas utilizadas para lograr esto se basan en valores puntuales como la cantidad de tareas en un *core* o su grado de inactividad (valores que podrían variar en el tiempo), y argumentablemente podrían ser optimizadas para manejar cargas dinámicas a través de la capacidad de adaptabilidad del ML.

2.2 Aprendizaje en un OS

Dirigiéndonos a los trabajos que proponen el uso de modelos predictivos y técnicas de aprendizaje en *kernel* se encuentra *Kernel Machine Learning* (KML) [25], un framework enfocado en la integración de algoritmos de ML en el kernel Linux, y es utilizado para implementar modelos de *Deep Learning* (DL) que buscan optimizar el ajuste de variables que definen el comportamiento de un sistema de almacenamiento en cuellos de botella . Los autores indican que los modelos de ML pueden atacar la compleja relación entre cargas de trabajo y parámetros ajustables a través de observación y adaptación en

tiempo real. Su trabajo logra: (i) mostrar que es posible integrar ML en sistemas operativos y en sistemas de almacenamiento, (ii) ofrecer flexibilidad a través de entrenamiento síncrono y asíncrono y en espacio de usuario, (iii) introducir la idea de APIs genéricos de ML expandibles en un futuro, (iv) aplicar KML al ajuste de los valores de *readahead* y *rsize* en el Network Filesystem (NFS) de Linux y (v) evaluar su solución utilizando múltiples cargas de trabajo complejas sobre dos dispositivos de almacenamiento diferentes (SATA-SSD y NVMe-SSD).

Los experimentos con KML muestran una mejora de hasta $\times 2.3$ el *throughput* de un escenario tradicional al ajustar dinámicamente el valor de *readahead* y una mejora de hasta $\times 15$ al hacer lo mismo con el valor de *rsize* de NFS. También obtuvieron resultados importantes sobre la utilización de recursos por parte del modelo de ML. Primero, un $0.18\,\%$ de overhead para el sondeo de recolección de datos en casos donde no hay ventanas de tiempo en la recolección (aumentar el tiempo de las ventanas reduce el overhead, pero consecuentemente reduce también la precisión de los modelos). Finalmente, un tiempo de $21\mu s$ en promedio para la ejecución de la predicción de una red neuronal (135 predicciones por segundo).

Los resultados de este trabajo fortalecen la idea de que la integración de modelos de ML no sólo brinda una mejor capacidad de optimización y ajuste a ciertos valores definidos tradicionalmente de manera estática en un sistema sino que también demuestran que, para escenarios específicos, es posible realizar las operaciones requeridas por uno de estos modelos sin generar pérdidas significativas. Algo que no se discute en este trabajo es la comparación de diferentes maneras de implementar ML en kernel, como el uso de aritmética de punto fijo o flotante, que podrían podría impactar en sus resultados.

Una investigación similar [1] hace énfasis en el problema de la falta de consideración de recursos de hardware de bajo nivel al momento de realizar decisiones con respecto al

balance de cargas en CFS . Los autores proponen como solución la implementación de un módulo de ML en kernel que logre: (i) tomar decisiones sobre migración de procesos simulando el comportamiento de CFS, (ii) incluir información de utilización de hardware de bajo nivel, (iii) entrenar al modelo para resolver casos de contención por alta demanda de estos recursos, (iv) aplicar aprendizaje reforzado para optimizar las decisiones de balance de cargas de trabajo. Su investigación se centró en la simulación del comportamiento de CFS (punto i) a través de un modelo entrenado para predecir las decisiones tomadas por CFS. Tras una recolección de datos realizada con la herramienta *Extended Berkeley Packet Filter* (eBPF), lograron obtener información sobre los parámetros de entrada y de salida de la función interna `can_migrate_task()`. El modelo fue entrenado de manera externa y se implementó dentro de una función alternativa a `can_migrate_task()` llamada `should_migrate_task()`. Los resultados muestran una precisión del 99 % al predecir decisiones del *kernel*, mientras que en términos de rendimiento el modelo presentó un 36 % de *overhead* para la función `can_migrate_task()` y de 13 % para la función `load_balance()`.

Este trabajo sugiere una buena capacidad de técnicas de ML para predecir comportamientos del *kernel*. El objetivo de la investigación estuvo relacionado principalmente a evaluar la precisión del modelo. Un tema importante que no se considera son las mejoras con respecto al rendimiento del sistema, y tampoco se muestra información acerca del efecto que tiene sobre el uso de memoria durante su ejecución. Tampoco se hace énfasis en la adaptabilidad del modelo frente a nuevas cargas de trabajo.

2.3 Implementación de ML en *kernel space*

Según lo visto hasta este punto, es argumentable que un sistema operativo puede beneficiarse del uso de técnicas de ML. Sin embargo parte importante de cada una de las investigaciones mencionadas se enfoca en la metodología para implementar estas

funcionalidades de manera que puedan ser usadas eficientemente por el sistema. Algunos trabajos previos han propuesto formas de inyectarlas de modo que algún subsistema de interés en su estudio (planificación de tareas, control de congestión en redes, etc) tenga acceso a ellas, ya sea: (a) implementando flujos de inferencia y entrenamiento como parte del *kernel*, (b) implementando flujos de inferencia y entrenamiento en espacio de usuario y brindando acceso a ellos desde espacio de *kernel* a través de algún mecanismo como eBPF, o (c) implementando flujos de inferencia en espacio de kernel y flujos de entrenamiento en espacio de usuario, comunicándose a través de algún mecanismo de acceso.

Este último método es implementado en dos trabajos que buscan extender la adaptabilidad de ciertos subsistemas del *kernel* de Linux a través de modelos predictivos basados en ML. LiteFlow [26], busca integrar el uso de *adaptive neural networks* para atacar problemas relacionados al manejo de funciones red con *OS kernel datapath*. Su enfoque se basa en la idea mencionada anteriormente que señala cómo los costos relacionados al mecanismo de implementación de este tipo de algoritmos puede comprometer las ventajas que traen.

Los autores argumentan en contra de las metodologías (a) y (b) mencionadas en el párrafo anterior, por diversas razones. Primero, al usar metodología (a), a pesar de que se aprovecha la gran madurez de la tecnología y herramientas para trabajar con redes neuronales en espacio de usuario (TensorFlow, PyTorch, etc), el costo de comunicación con el espacio de kernel es muy alto. Otros trabajos intentan reducir este costo, pero los autores de LiteFlow afirman, después de haber realizado experimentos para encontrar intervalos de comunicación óptimos, que no pudieron encontrar una configuración que no degrade el rendimiento de la función, perjudicando las ventajas que trae el uso de la red neuronal. Segundo, al usar la metodología (b), la cantidad de recursos requeridos para el ajuste del modelo interfiere con la lógica de los flujos del *datapath* al ser implementado en espacio de *kernel*, y a pesar de que el uso de operaciones de punto flotante alcanza una mejor

precisión, este genera un *overhead* adicional.

Debido a ambos argumentos, LiteFlow opta por una solución híbrida que separa las funciones de inferencia y entrenamiento en dos módulos, siendo la inferencia ejecutada en espacio de *kernel* y el entrenamiento en espacio de usuario. Lograron esto a través del ajuste del modelo sólo cuando se cumplen ciertas condiciones (usualmente cuando el modelo no tiene un buen rendimiento para la carga de trabajo en un punto de tiempo definido), y se evita generar un cuello de botella al sobrescribir los parámetros del modelo (dado que más de una función puede solicitar el uso del modelo) teniendo dos copias de este en todo momento, una activa y una en espera. Para ajustar el modelo, LiteFlow permite a los usuarios recolectar datos a través de funciones en el *kernel datapath*, que verifican la coherencia de los datos y la red neuronal.

Probaron utilizar LiteFlow en experimentos con el objetivo de evaluar su rendimiento en el manejo de congestión de red. Para esto generan dos modelos previamente propuestos llamados Aurora [27] y MOCC [28] con dos objetivos: (i) comparar el rendimiento de LiteFlow utilizando estos modelos (LF-Aurora y LF-MOCC) con sus versiones en espacio de usuario (CCP-Aurora y CCP-MOCC), y (ii) comparar su rendimiento con un algoritmo tradicional de *kernel* llamado CUBIC [29]. Los resultados muestran mejoras en el *goodput* de LiteFlow con respecto a las implementaciones en espacio de usuario de Aurora y MOCC, de 44.4 % y 26.6 %, respectivamente y, también se observa una mejora en el *overhead* con respecto a CUBIC, con una reducción del 17.5 %. Experimentos similares fueron realizados para evaluar LiteFlow en contextos de planificación de flujos y balance de cargas de tráfico comparando modelos integrados con LiteFlow contra los mismos modelos implementados a través de un *char device*. Se observó una mejora considerable de rendimiento en ambos escenarios, donde en el de balance de cargas, LiteFlow tuvo una reducción del 34.3 % de tiempo de ejecución en flujos cortos y de un 56.7 % en flujos largos. En lo que corresponde a la planificación de flujos, se observó una mejora

con respecto al tiempo de ejecución de flujos de un 10.9 % para flujos cortos y de 33.7 % para flujos largos.

De manera similar, el segundo trabajo propone el *reconfigurable kernel datapath* [30], un método que busca utilizar *reconfigurable match tables* (RMTs) como interfaz para el acceso al servicio de uno o varios modelos de ML. Similarmente a los autores de LiteFlow, los autores de este trabajo argumentan que debido a que el sistema operativo soporta múltiples tipos de aplicaciones en múltiples plataformas, éste debe ser capaz de adaptarse a escenarios específicos para no perder rendimiento, y alcanzar una capacidad generalizada para escenarios no vistos previamente. Con este objetivo en mente, utilizan RMTs para integrar el manejo de redes neuronales de manera que cada entrada de la tabla (representada por una serie de cabeceras de paquetes de red) representa un escenario de ejecución, y cada acción tomada en base a estos campos puede modificar el contexto de ejecución o llamar a un modelo de ML. Las políticas son controladas por una API de control, permitiendo agregar, eliminar y modificar entradas y modelos, logrando desacoplar las funcionalidades de inferencia y entrenamiento.

Validaron la efectividad de este método realizando experimentos con dos casos de estudio: (i) *page prefetching* utilizando un árbol de decisión para predecir qué páginas de memoria serán solicitadas próximamente, y (ii) predicción de decisiones de migración de CFS. En el primer caso, el método propuesto muestra mejoras de precisión del 28 %-80 % sobre la predicción de páginas que serán solicitadas con respecto a la heurística por defecto del *kernel* y del 23 %-44 % con respecto al modelo Leap [31]. En el segundo caso de estudio, se enfocaron en uno de los trabajos previamente mencionados que utiliza un MLP para predecir decisiones de migración de CFS [1] y replican sus experimentos utilizando el método propuesto. Similarmente al trabajo mencionado, alcanzan una precisión del 99 % al predecir decisiones de CFS, y reducen la complejidad de la red utilizando sólo los parámetros de mayor correlación con la salida, haciendo un intercambio de precisión

obteniendo un 94 % para reducir el tiempo de computación.

Título	Contribución	Método de Implementación	Resultados	Diferencias
Machine Learning for Load Balancing in the Linux Kernel[1]	MLP para predicción de decisiones de CFS	En kernel, configuración opcional de compilación	Alta precisión al predecir decisiones de CFS, alto overhead	No busca mejorar el rendimiento del sistema, no se experimenta con diversas cargas de trabajo
Performance Improvement of Linux CPU Scheduler Using Policy Gradient Reinforcement Learning for Android Smartphones[16]	Learning EAS, Reinforcement Learning para ajuste de variables TARGET_LOAD y sched_migration_cost en scheduler y gobernador de frecuencias de dispositivos móviles	En kernel, como una librería	Mejoras en rendimiento y consumo energético con respecto a EAS para aplicaciones móviles	No especifica métodos de implementación. Se concentra en resolver problemas de optimización y no necesariamente de adaptabilidad
OS Scheduling with NEST[17]	NEST Scheduler,nuevas heurística para planificación de procesos	En kernel, modificación del scheduler	Speedup significativo y reducción del consumo energético en scripts de configuración y aplicaciones multithread	No se utiliza ML
Machine Learning framework to improve storage system performance	KML, modelos para optimización de ajuste de las variables rsize y readahead en sistemas de almacenamiento	Inferencia como módulo de kernel, entrenamiento como programa de usuario	Alta precisión para la predicción	ML orientado a sistemas de almacenamiento
LiteFlow: Towards High Performance Adaptive Neural Networks for Kernel[26] Datapath	LiteFlow, implementación eficiente de modelos de ML orientados a manejo de flujos	Inferencias en espacio de kernel, entrenamiento y ajuste de los modelos desde espacio de usuario	Mejoras considerables en rendimiento en control de congestión, planificación de flujos y balance de tráfico	ML orientado a tráfico de redes
Toward Reconfigurable Kernel Datapaths with Learned Optimization[30]	Reconfigurable Kernel Datapath, método basado en RMTs para integración de ML en kernel de manera eficiente.	Inferencias en espacio de kernel utilizando RMTs, API de control para espacio de usuario	Alta precisión en predicción de páginas a ser leídas (readahead), alta precisión en predicción de decisiones de CFS	Utiliza RMTs para implementación

TABLA 2.1: Resumen de cada referencia, señalando contribuciones principales, resultados y diferencias en comparación con este trabajo.

Capítulo 3

MARCO TEÓRICO

Este capítulo define conceptos fundamentales sobre el desarrollo de *kernel* y ML.

3.1 Kernel Linux

Linux es un *kernel* basado en el sistema operativo *Unix*, que soporta el estándar *Portable Operating System Interface* (POSIX) [32]. Cuenta con las funcionalidades de un sistema *Unix* incluyendo multiprocesamiento, memoria virtual y manejo de redes multicapa, entre otras capacidades [15]. Siendo un *kernel* monolítico, Linux es ejecutado como un sólo programa en una partición específica de memoria virtual mapeada a la memoria física del sistema, conocida como espacio de *kernel* o *kernelspace*. La partición restante que incluye al resto de direcciones del sistema se encuentra disponible para programas de usuario, y es conocida como espacio de usuario o *userspace*. Linux es uno de los *kernel* más utilizados globalmente, siendo el sistema utilizado en el 100 % de las 500 supercomputadoras más rápidas del mundo en el 2023 [33] y el núcleo del OS Android, que potencia el 85 % de los smartphones en el mercado a nivel mundial [34].
Además, al ser un *kernel* de código abierto, Linux brinda una gran facilidad de aprendizaje y colaboración contando con un gran número de desarrolladores, una extensa documentación y diversas interfaces para desarrollar y experimentar con componentes adicionales. Por estas razones, existe una gran cantidad de investigaciones realizadas sobre *Linux* en la actualidad, y la mayoría de trabajos que se orientan al uso de ML en la optimización de sistemas operativos utilizan *Linux* como herramienta y ambiente de estudio.

3.2 CFS Scheduler

Linux soluciona el problema de CMP a través de un módulo llamado *load balancer*. Este forma parte de CFS, el *scheduler* por defecto introducido en la versión 2.6.23 de *Linux* y que desde entonces ha sido utilizado para la multiplexación de tareas generales. Según se menciona en la documentación oficial [15], el diseño de CFS puede ser resumido en una oración: "CFS modela un *multitasking* ideal y preciso en hardware real".
Para lograr acercarse a este "*multitasking* ideal", CFS introduce el concepto de *virtual runtime*, una medida que representa el tiempo de ejecución total que una tarea ha tenido en un CPU. Cada proceso cuenta con un valor de `vruntime`, haciendo posible el monitoreo de tiempo en CPU esperado para cada tarea. Adicionalmente se aplica el concepto de prioridad al *virtual runtime*, funcionando como factor de escala al momento en que su valor es calculado, por lo que la prioridad de un proceso determina también cuánto tiempo se le asigna en CPU. En general, se puede atribuir el comportamiento de CFS a las siguientes tres cláusulas:

1. El *scheduler* siempre asigna el CPU a la tarea cuyo valor de *vruntime* sea el mínimo en la cola de ejecución respectiva.

2. El tiempo en CPU de cada tarea es acumulado a su valor de `vruntime` en cada llamada a la función `schedule()`, y la velocidad a la que se acumula depende de la prioridad de la tarea.

3. A cada tarea nueva que es encolada se le asigna el `vruntime` mínimo local (de la cola de ejecución).

CFS utiliza una cola de ejecución para cada *core*, y cada una de estas colas hace uso de un *Red Black Tree* para mantener la estructura y orden de sus tareas correspondientes basándose en el `vruntime` como llave. De esta manera, la tarea con el menor valor de `vruntime` se encuentra siempre a la izquierda del árbol y puede ser asignada en tiempo

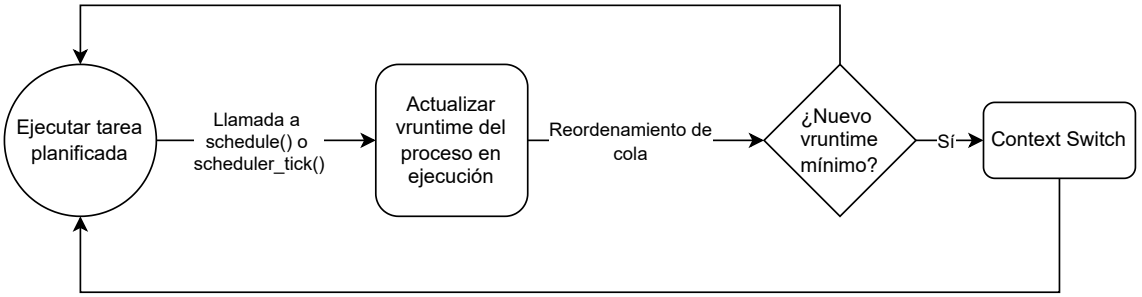


FIGURA 3.1: Diagrama del flujo de CFS. Las tareas se ejecutan hasta una llamada al *scheduler*, que actualiza los *vruntime* y elije una nueva tarea

constante. El comportamiento de este *scheduler* se encuentra resumido en la Figura 3.1. Como se explica en este diagrama las tareas se ejecutan por un período de tiempo que termina cuando se llama al *scheduler*, luego se actualizan los valores de `vruntime` y se asigna el CPU a la siguiente tarea que tenga el nuevo mínimo local.

Para asegurar que exista un balance entre las cargas de trabajo de los *cores*, CFS asigna un peso, o *load*, a cada proceso, representando su uso promedio del CPU. Cada cola de ejecución mantiene una noción de su carga de trabajo total, y es este mismo valor el que se utiliza para identificar situaciones donde es necesario hacer una llamada al *load balancer*. Este módulo se encarga de identificar cuáles colas de ejecución se encuentran inactivas o sobrecargadas, y es invocado sólo en una de las siguientes situaciones [15]: (i) al crear un nuevo proceso, (ii) al despertar un proceso dormido, (iii) periódicamente a través de la función `scheduler_tick()` .

El proceso de identificación y balance de cargas de trabajo se encuentra brevemente resumido en el Algoritmo 1. Este itera sobre todos los grupos de CPUs que contienen al CPU actual y verifica si su intervalo de balance ha expirado. De ser el caso, se llama a la función `load_balance()`, la cual está resumida entre las líneas 4 y 8. Esta función realiza lo siguiente: busca el grupo de CPUs con mayor carga de trabajo, bloquea su cola de ejecución y la cola de ejecución actual, y finalmente empieza a reubicar procesos de una cola haca otra.

Algoritmo 1: *Load balancer* de CFS

Entrada: CPU actual y su estado de inactividad

```
1  para todo D | cur_cpu ∈ D hacer
2      si intervalo de rebalance expira entonces
3          busiest_grp ← find_busiest_group(D)
4          si busiest_grp entonces
5              grp_rq ← busiest_grp.rq
6              lock(cur_cpu.rq)
7              lock(busiest_grp_rq)
8              move_tasks(cur_cpu.rq, grp_rq)
9          fin
10     fin
11 fin
12
```

3.3 eBPF y *kprobes*

El rastreo, o *tracing*, de *kernel* es un método eficiente y robusto orientado al *debugging* de sistemas complejos [35]. Existen diversas herramientas que utilizan varios métodos para lograr este objetivo y entre ellas se encuentra eBPF, una tecnología que permite crear programas controlados por eventos que se ejecutan dentro del mismo sistema de manera segura y eficiente [36].

Un programa eBPF puede estar enlazado a un *hook*; un punto de ejecución específico del sistema (funciones, *system calls*, etc.), y si se desea enlazar un programa a una función que no está definida como *hook* predeterminado, eBPF permite crear *kprobes* para instalar programas en casi cualquier parte del *kernel* o en aplicaciones de usuario. Un *kprobe* permite irrumpir cualquier rutina del kernel para recolectar información de manera no-disruptiva, y puede invocar instrucciones *trap* en casi cualquier dirección del *kernel* especificando un controlador o *handler* específico para cada evento [15]. En la Figura 3.2 se explica la lógica de un programa de eBPF. En este caso, cada vez que `execve()` retorna, se invoca a la función `syscall_ret_execve()` que almacena información sobre el ID, el tipo de retorno y el nombre del proceso y lo almacena en un *buffer* accesible desde espacio de

usuario.

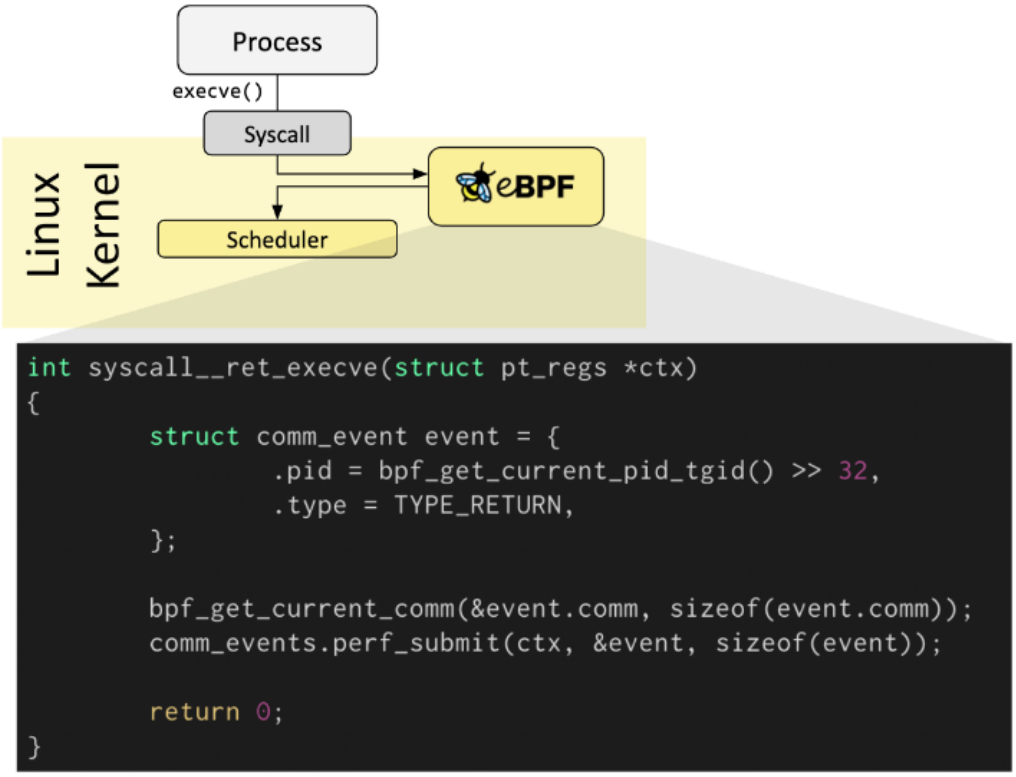


FIGURA 3.2: Ejemplo de un programa eBPF enlazado a la *system call* `execve()` [36]. Este programa obtiene datos sobre el proceso y los almacena en una estructura de datos nativa de eBPF.

3.3.1 Operaciones de punto flotante en en *Kernel*

Los programas de usuario suelen realizar operaciones sobre valores representados en notación de punto flotante. Esto es posible gracias a las abstracciones del sistema operativo o a través del uso de instrucciones especiales que se encargan de proveer los mecanismos para su ejecución. Existen procesadores que cuentan con una unidad lógica

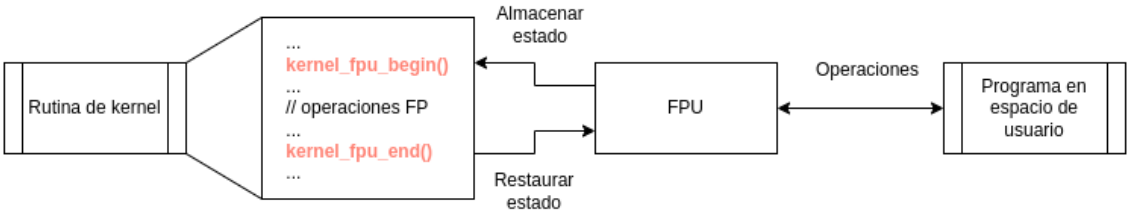


FIGURA 3.3: Ejemplo de una llamada a las rutinas `kernel_fpu`. Este diagrama representa únicamente las interacciones con el FPU desde una perspectiva de espacio de *kernel* y usuario, no representa una arquitectura ni diseño del sistema

llamada *Floating-Point Unit* (FPU) que se encarga de realizar operaciones de punto flotante. A pesar de esto, los desarrolladores de Linux optan por (y recomiendan fuertemente) no utilizar operaciones de punto flotante en espacio de *kernel*. Esto sucede por tres razones principales: 1) en sistemas con un FPU, el *kernel* no almacena normalmente el estado de los registros de esta unidad en cada rutina, ya que estos son accesados por el espacio de usuario constantemente, 2) muchas arquitecturas no cuentan con soporte para este tipo de operaciones, y 3) incluso si lo hicieran, almacenar y restaurar los registros del FPU en cada llamada al *kernel* sería costoso [37]. A pesar de esto, Linux cuenta con la implementación de una serie de rutinas para las arquitecturas específicas x86 que permiten hacer uso del FPU desactivando el *preemption* y almacenando el estado de sus registros. Estas rutinas son `kernel_fpu_begin()` y `kernel_fpu_end()` [38], y están diseñadas para usos especiales en secciones de código no bloqueante. Como se observa en la Figura 3.3, cuando se llama a `kernel_fpu_begin()` se almacenan los registros del FPU, que contiene el estado de las operaciones de un proceso en espacio de usuario. Luego, al llamar a `kernel_fpu_end()`, se restaura el estado de estos registros. Es importante resaltar que el *preemption* se encuentra desactivado durante este intervalo de ejecución, descartando la posibilidad de inconsistencia en los registros.

3.3.2 Migraciones forzadas

El *load balancer* debe revisar si es preferible que una tarea sea migrada, o si debe permanecer en el mismo *core*. Con este fin se llama a la función `can_migrate_task()` que retorna un valor de 0 si la tarea debe permanecer y 1 si la tarea debe ser migrada. Esta función se basa en heurísticas fijas para tomar decisiones; por ejemplo, si una tarea se encuentra en ejecución o si se encuentra activa o "caliente" en *cache*, no debe ser migrada. Dentro de estos escenarios, puede darse el caso de que ya hayan ocurrido muchos intentos fallidos de balance o que la tarea se encuentre asignada a otro nodo mientras está caliente en cache. Frente a esta situación, CFS decide migrar la tarea registrando en las estadísticas que ocurrió esta migración forzada. En el peor de los casos, esto sería perjudicial por dos razones: (i) una serie de balances fallidos previos a una migración forzada resultaría en llamadas innecesarias a esta rutina, y (ii) dependiendo de la topología, migrar una tarea caliente en *cache* signigica que las lineas de *cache* terminarían cargadas con información de un proceso que ya no se ejecuta en aquel grupo de cores. En este trabajo se argumenta que el escenario descrito podría ser evitado si se logra predecir su ocurrencia en base a información del contexto de ejecución como la cantidad de migraciones previas, qué CPUs se encuentran en migración, la carga de trabajo de las colas y la inactividad de los CPUs. La función `can_migrate_task()` se encuentra resumida en el algoritmo 2, con énfasis en el escenario mencionado. Si el CPU actual no está dentro de los CPUs permitidos para ejecutar el proceso o si el proceso se encuentra en ejecución, no se realiza la migración. Una vez revisadas estas condiciones, si el proceso se encuentra frío en *cache*, se encuentra asignado a otro nodo NUMA o si ya se excedió la cantidad límite de intentos fallidos de balance, se migra el proceso. Nótese que el contador de migraciones forzadas aumenta cuando se migra una tarea a pesar de estar caliente en *cache*.

Algoritmo 2: Función `can_migrate_task()` de CFS

Entrada: Contexto de balance lb_env y proceso en ejecución p
Resultado: Decisión de migración

```
si throttled_lb_pair OR cpu_not_allowed OR task_running entonces
    return 0
fin
task_cache_hot ← task_hot(p, lb_env)
si NOT task_cache_hot OR balance_failed > nice_tries entonces
    si task_cache_hot entonces
        forced_migrations ← forced_migrations + 1
    fin
    return 1
fin
return 0
```

3.4 Redes Neuronales

Los *multilayer perceptron* (MLP), también conocidos como redes neuronales prealimentadas, representan modelos que comprenden múltiples capas modulares de regresión logística [39]. Estos modelos se utilizan para la predicción o inferencia de algun evento o variable en base a transformaciones no lineales de sus entradas. Sus casos de uso y aplicaciones reales incluyen la identificación y clasificación de tumores [40], o el reconocimiento facial [41]. Cada una de las capas de una red aplica funciones no lineales sobre una combinación lineal de sus entradas, de manera que el modelo busca ajustar sus parámetros $\{w_j\}$ para cada una de las unidades (o neuronas) a través de una etapa de entrenamiento. Entonces, como se argumenta en la fuente mencionada, podemos describir a un MLP como una serie de transformaciones funcionales.

La figura 3.4 muestra la arquitectura de un MLP. Como se observa, el modelo consiste en una capa de entrada, una serie de L capas ocultas, y una capa de salida. Cada capa l consiste en un número M (correspondiente al número de salidas) de combinaciones lineales de las N valores de entrada $x_1, \ldots, x_N$, transformados por una función de activación

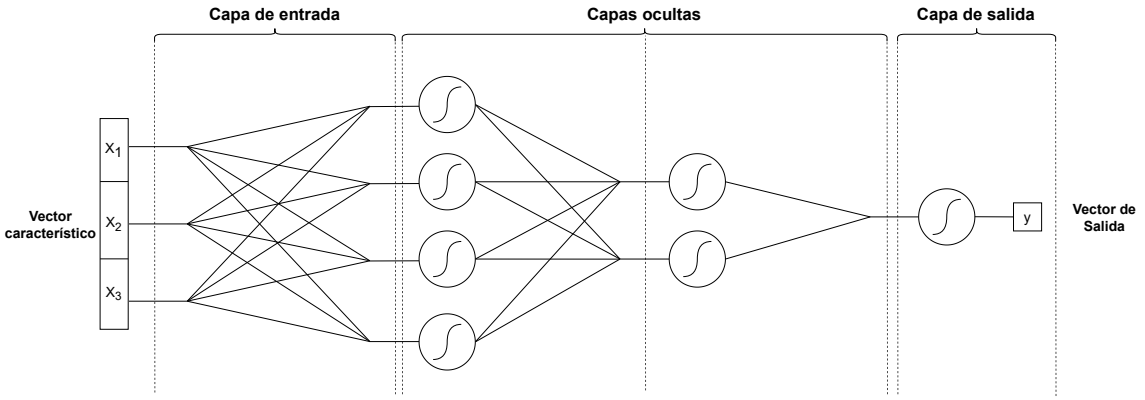


FIGURA 3.4: Diagrama de una red neuronal con dos capas ocultas, un vector de entrada de tres dimensiones y un sólo valor de salida.

$A(x)$ no lineal (representada por un círculo o nodo en el diagrama), y luego multiplicados por un peso w_{ij} donde ij indica el paso de la entrada i a la salida j. El resultado $o_j^l = x_j^{l+1}$ corresponde al valor de la entrada j de la siguiente capa $l+1$. Entonces:

$$o_j^l = x_j^{l+1} = \sum_{i=1}^{N} A(x_i^l) w_{ij}^l \tag{3.1}$$

Similarmente, las entradas de la siguiente capa $l+2$ pueden ser calculadas:

$$o_j^{l+1} = x_j^{l+2} = \sum_{i=1}^{N} A(x_i^{l+1}) w_{ij}^{l+1} = \sum_{i=1}^{N} A(\sum_{i=1}^{N} A(x_i^l) w_{ij}^l) w_{ij}^{l+1} \tag{3.2}$$

Lo cuál se explica gráficamente en la figura 3.5. Este proceso se repite en cada capa hasta obtener el vector de salida y. Las funciones de activación $A(x)$ utilizadas suelen ser sigmoidales, lo cuál facilita la tarea de clasificación, y pueden variar entre distintas arquitecturas. El proceso de generación del vector de salida de la red y a partir del vector de entrada x a través de estas transformaciones se conoce como *forward propagation*.

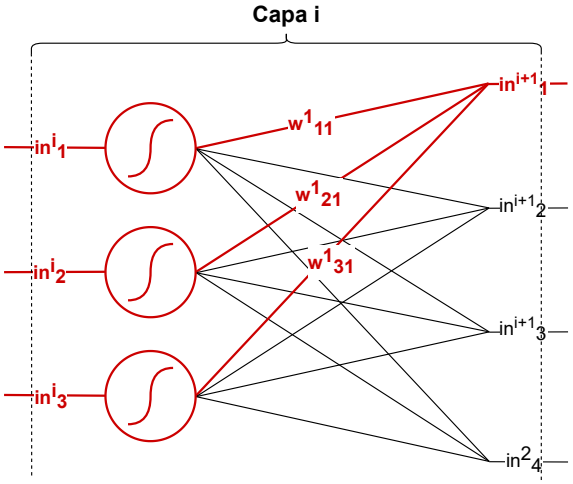


FIGURA 3.5: Diagrama de una capa interna. Nótese que la salida representa la entrada de la siguiente capa. Cada salida x_j^{l+1} está dada por $\sum_{i=1}^{N} A(x_i^l) w_{ij}^l$.

Capítulo 4

MÉTODOS

En este capítulo se explica a detalle la metodología de implementación con las herramientas necesarias para la implementación de nuestro modelo.

Para predecir la ocurrencia de migraciones forzadas *cache-hot*, se propone el uso de una red neuronal que recibe parámetros relacionados al contexto de la migración de tareas y predice la probabilidad de que ocurra una migración de este tipo, pudiendo de esta manera ser migrada sin necesidad de evaluarse nuevamente repetidas veces, e idealmente migrándola en un instante de tiempo en el que sea menos costoso. Además este trabajo busca implementar una librería híbrida que permita entrenar modelos en espacio de usuario e inyectarlos al *kernel* de manera eficiente, y consecuentemente evaluar el rendimiento y consumo de recursos que trae consigo. La figura 4.1 muestra la arquitectura de la librería, donde al lado derecho se encuentran los componentes que corren en espacio de usuario, y a la izquiera los que corren en espacio de *kernel*

Nuestra metodología se basa fuertemente en la investigación de Chen et al. [1]; se utiliza un modelo similar y la recolección de datos se realiza de la manera propuesta en aquel trabajo. Nuestro trabajo, al igual que [1] estudian la idea de predecir las decisiones de migraciones de CFS. Sin embargo, la investigación descrita en este documento se diferencia en dos puntos clave:

- Se extiende el objetivo de predicción, prediciendo las ocurrencias de migraciones forzadas para migrar las tareas antes de que ocurran. Este último punto surge debido

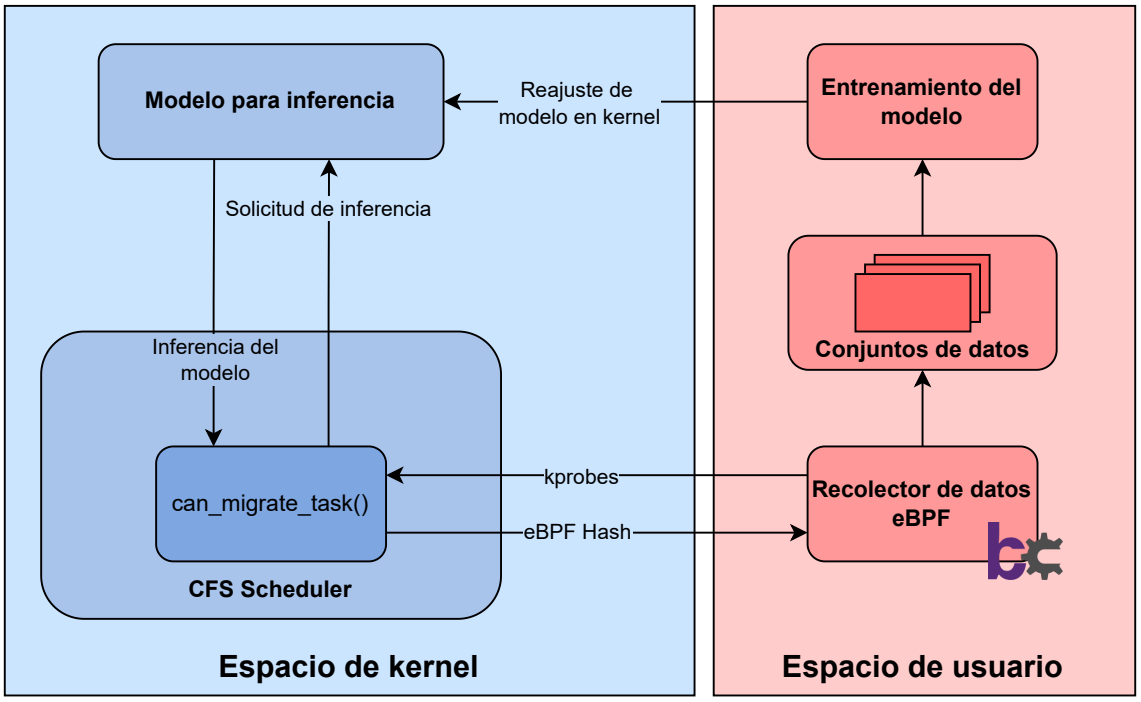


FIGURA 4.1: Estructura de la librería propuesta para la integración de ML en *kernel*. Todos los pasos relacionados al entrenamiento son ejecutados en espacio de usuario, mientras que una iteración de modelo resultante realiza inferencias en espacio de *kernel*

al alcance del trabajo mencionado, que logra predecir con alta precisión las decisiones de CFS, lo cual significa que en el mejor escenario el modelo se dedicará a replicar las decisiones que CFS ya es capaz de tomar.

- Se extiende [1] mediante la integración de la perspectiva de adaptabilidad abordada en LiteFlow y *Reconfigurable kernel datapath* [26, 30] a su modelo en *kernel* creando una serie de programas y herramientas que permitan actualizarlo y reconfigurarlo a través de programas en *userspace*.

4.1 Implementación de Librería de ML

A continuación se explica detalladamente la implementación de LibML, una librería de ML híbrida espacio para uso en espacio de usuario y *kernel*. Resaltamos que el

código que el *kernel* permite ejecutar es limitado con respecto al espacio de usuario. Desarrollar redes neuronales en *kernel* trae grandes dificultades por tres razones principales: (i) falta de soporte para librerías (matemática, lectura/escritura de archivos, *etc*), (ii) falta de soporte para operaciones de punto flotante, y (iii) implementación de algoritmos de inferencia y entrenamiento desde cero (*forward/back-propagation, cálculo de gradientes, etc*).

4.1.1 Recolección de datos

Se utiliza la herramienta BPF Compiler Collection (BCC) con el objetivo de obtener información del estado del sistema en tiempo real. BCC maneja un *front-end* para el uso de varias herramientas del *kernel*, entre ellas *kprobes*, a través de eBPF. Se instalan *kprobes* en la entrada y salida de alguna función de interés, en este caso la función `can_migrate_task()`, y de esta manera se pueden obtener sus parámetros de entrada y sus valores de salida. En la Figura 4.2 se representa gráficamente la instalación de *kprobes*, y cómo son utilizados para generar conjuntos de datos. Como se observa aquí, se instala un *kprobe* en la entrada de la función para obtener sus argumentos de entrada y un *kretprobe* a la salida para obtener sus valores de retorno. Esta configuración de *tracing* es epecífica para los objetivos de esta investigación, y puede ser modificada fácilmente para otros fines, como por ejemplo la recolección de parámetros de un *system call*. La instalación de *kprobes* puede ocurrir en cualquier instante de la ejecución del *kernel*, permitiendo la recolección de datos en cualquier escenario de manera aislada.

Para poder identificar las migraciones forzadas, realizamos una modificación al código de la función `can_migrate_task()` para que se retorne un valor de 2 cuando ocurra una migración forzada (siendo por defecto 1 cuando ocurre una migración). Esta modificación se observa en el Listado 1. Aquí se puede observar el valor de retorno de 2 dentro de la cláusula `if(task_cache_hot == 1)`, cuya condición sólo se cumple

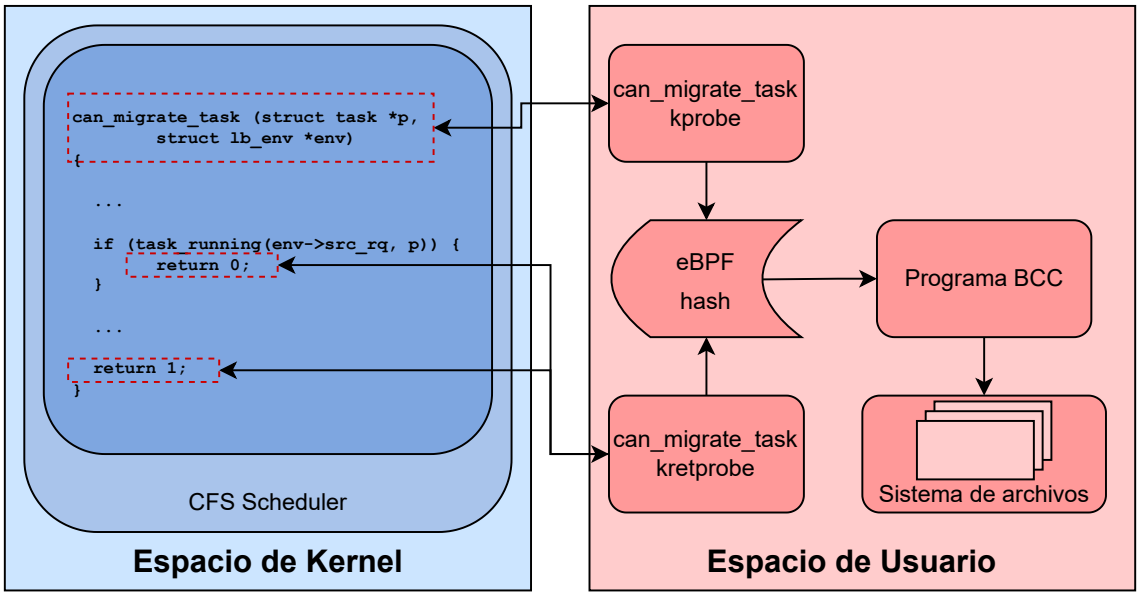


FIGURA 4.2: Vista detallada del proceso de recolección de datos a través de *kernel tracing*. En este caso se instalaron *kprobes* en `can_migrate_task()`

cuando una tarea *cache-hot* está siendo migrada. Una vez recolectados los datos, identificamos los procesos que realizaron una migración agresiva para luego obtener su i) PID, ii) CPU de origen y iii) CPU de destino y iv) la cantidad de migraciones que han realizado hasta ese punto. Con esta información se puede agrupar series de llamadas a la función `can_migrate_trace()` realizadas por el mismo proceso, desde el mismo CPU de origen hacia el mismo CPU de destino y con la misma cantidad de migraciones realizadas hasta el momento (indicando que el *load balancer* ha evaluado la migración de esta tarea en repetidas ocasiones), representando un historial de llamadas previas a una migración agresiva y así saber con certeza qué migraciones pudieron haber sido forzadas antes de tiempo con un mismo resultado. De esta manera, el conjunto de datos cuenta con los atributos listados en la Tabla 5.3 cumpliendo la función de vectores característicos, y con una etiqueta categórica que representa la observación eventual de una migración forzada, funcionando como vector de salida. La Figura 4.3 muestra el proceso de pre-procesamiento realizado hasta este punto. Como se observa aquí, los registros con un valor de migración igual a 2 son agrupados junto con sus llamadas previas, para luego marcarlos en el

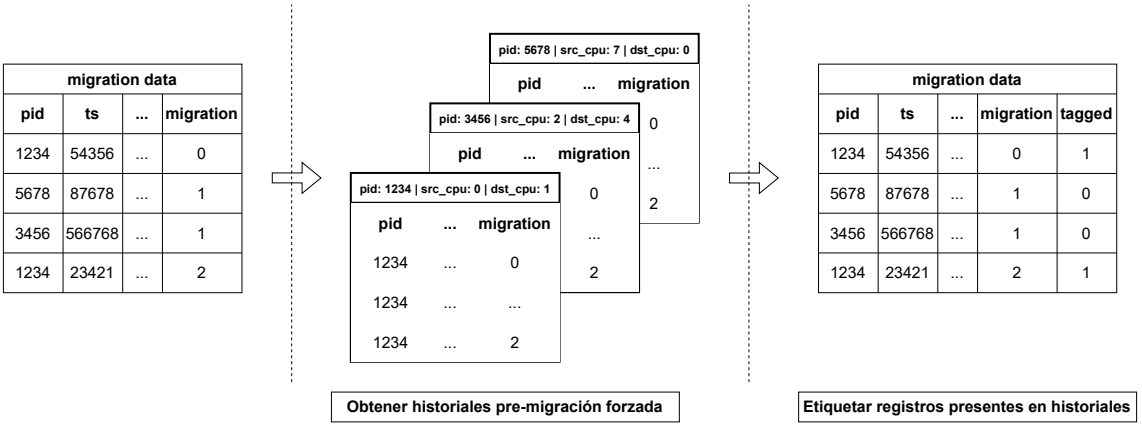


FIGURA 4.3: Proceso de obtención de historias de migraciones agresivas y obtención de registros etiquetados.

conjunto de datos original. Después, se limpian los registros a través de los siguientes pasos:

1. Conversión de los datos a tipo flotante
2. Balance de representatividad entre ambas categorías (50/50)
3. Exclusión de campos sin varianza
4. Exclusión de registros con valores inválidos
5. Normalización de datos a nivel de columna

4.1.2 Entrenamiento del modelo

El entrenamiento del modelo es realizado en espacio de usuario. La librería implementada permite la construcción de redes con arquitecturas pequeñas (con asignaciones de memoria no más grandes que el tamaño de página), un manejo limitado de conjuntos de datos, el entrenamiento de modelos, algunas funciones matemáticas y las validaciones correspondientes. El alcance del manejo de conjuntos de datos actualmente sólo permite

```
static
int can_migrate_task (struct task_struct *p, struct lb_env *env)
{
      ...
      /* Aggressive migration if:
      * 1) destination numa is preferred
      * 2) task is cache cold, or
      * 3) too many balance attempts have failed.
      */
     tsk_cache_hot = migrate_degrades_locality(p, env);
     if (tsk_cache_hot == -1)
         tsk_cache_hot = task_hot(p, env);

     if (tsk_cache_hot <= 0 ||
         env->sd->nr_balance_failed > env->sd->cache_nice_tries) {
         if (tsk_cache_hot == 1) {
             schedstat_inc(env->sd->lb_hot_gained[env->idle]);
             schedstat_inc(p->se.statistics.nr_forced_migrations);
             return 2;       // Linea agregada para recoleccion
         }
         return 1;
     }

     schedstat_inc(p->se.statistics.nr_failed_migrations_hot);
     return 0;
}
```

LISTING 1: Sección de código de la función `can_migrate_task` modificada para la recolección de datos (incluye código de [23])

la normalización y partición como funciones de tranformación; cualquier otro preprocesamiento debe ser realizado de manera externa.

Para reflejar el entrenamiento en espacio de *kernel*, el programa de usuario utiliza una *system call* especial que recibe como argumento los nuevos parámetros del modelo y se encarga de actualizarlos en el *kernel*. El flujo se encuentra detallado en la Figura 4.4. De esta manera se vuelve posible reconfigurar el modelo en base a cualquier conjunto de datos que se haya recolectado.

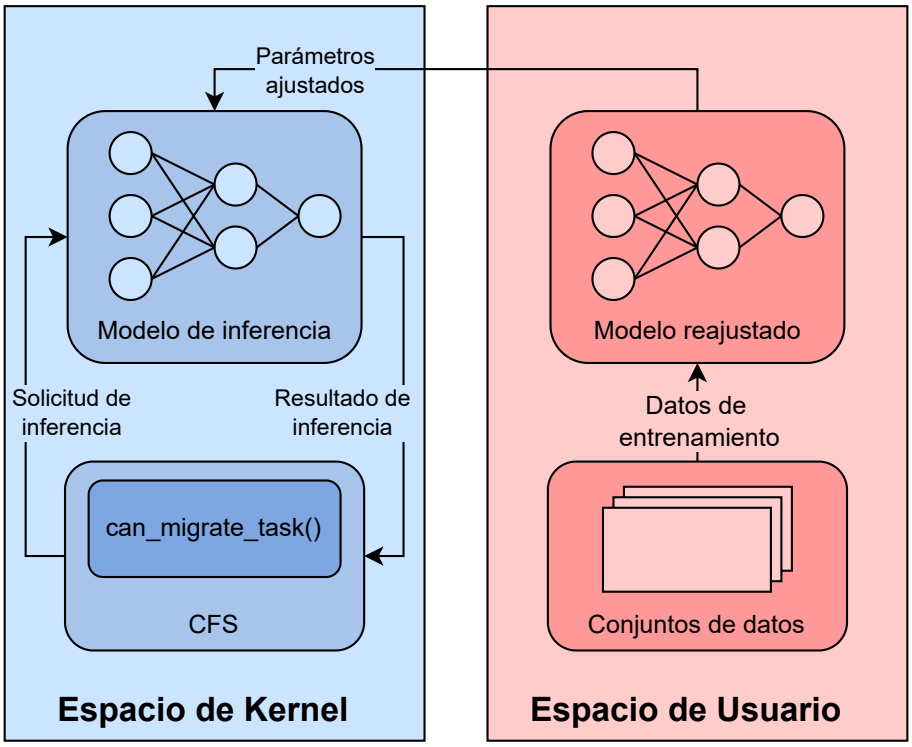


FIGURA 4.4: Metodología de trabajo orientada a la implementación y evaluación del modelo en el *kernel* de *Linux*

4.1.3 Implementación de inferencias en Kernel

En el caso del problema de migraciones forzadas, podemos tratar $y = f(x)$ como el resultado de la función `can_migrate_task()`, o en otras palabras, la decisión de migración de CFS. Por otro lado, podemos deconstruir las variables p y env mencionadas en el Algoritmo 2 para obtener un vector característico x. Esto permite que, a través de un aprendizaje supervisado basado en datos recolectados, se pueda ajustar los parámetros de la función $f(x)$ para que se cumpla $y = f(x)$ al realizar predicciones con el modelo entrenado. Entonces, tendríamos que la cardinalidad n del vector característico estaría dada por la cantidad de variables que se decida utilizar para predecir las migraciones, y el modelo tendría n entradas y una cantidad l de capas internas.

El objetivo de la implementación del modelo es reemplazar el flujo tradicional de

la función `can_migrate_task()` por un nuevo flujo que haga uso de las inferencias realizadas por la red neuronal para tomar decisiones de migraciones. Este se encuentra descrito en el Algoritmo 3. Como se observa, a excepción de los casos donde se debe evitar estríctamente migrar una tarea, toda la lógica de evaluación de migraciones es reemplazada por la inferencia del modelo. Esta inferencia recibe un vector que es generado en la línea 4 a partir de las variables `p` y `lb_env`, y contiene los atributos característicos que sirven de entrada para el modelo. Para integrar la funcionalidad de la red en el *load balancer* de *Linux*, es necesario implementar el modelo desde cero en el lenguaje C, haciendo uso de las rutinas exclusivas del *kernel* para realizar asignaciones de memoria y de funciones de aproximación para algunas operaciones matemáticas. Este enfoque, como se comenta en [26], "...introduce un desarrollo y depuración dramáticos para redes neuronales al tener que utilizar lenguajes de programación de sistemas como C y enfrentarse a restricciones *e.g*, soporte limitado de librerías, en espacio de *kernel*". A continuación se explica cómo se abordaron estos problemas.

Algoritmo 3: Nuevo flujo de `can_migrate_task()` con ML

Entrada: Contexto de balance lb_env y proceso en ejecución p
Resultado: Decisión de migración

```
1 si throttled_lb_pair OR cpu_not_allowed OR task_running entonces
2 |   return 0
3 fin
4 in ← generate_input(p, lb_env)
5 return inference(in)
```

4.1.4 Uso de memoria

Reservar memoria para un proceso en espacio de *kernel* no es tan simple como hacerlo en un programa de usuario. En primer lugar, no se cuenta con la librería estándar para realizar este tipo de funciones, por lo que la familia de funciones `kmalloc()` [15] sirve como conjunto de herramientas análogo para la asignación de memoria de la mayoría de vectores y matrices utilizados por la red neuronal. Afortunadamente, debido a que

se utilizan dimensiones pequeñas para la red (por temas de alcance, y también porque utilizar redes extensas en *kernel* sería contraproducente), no es necesario asignar bloques de memoria más grandes que el *page size*. En segundo lugar, se debe tener en cuenta el contexto de una llamada a `kmalloc()` para seleccionar una *flag* apropiada, e.g. el uso de `GFP_ATOMIC` dentro de un contexto de *interrupt* es apropiado ya que no permite *sleeps*, a diferencia de otras *flags*. Adicionalmente, es posible mantener una noción del uso de memoria del modelo haciendo uso de la función `ksize()` [15], que retorna el tamaño en *bytes* de un puntero asignado con alguna de las funciones mencionadas.

4.1.5 Operaciones de punto flotante

Se utilizaron las rutinas `kernel_fpu_begin()` [23] y `kernel_fpu_end()` para permitir el uso de operaciones de punto flotante en el *kernel*. Esta facilidad puede ser utilizada para implementar las operaciones que el modelo utiliza para las funciones de inferencia, reduciendo drásticamente la dificultad de implementación a cambio de un ligero *overhead*.

La compilación de un *kernel* modificado para el uso de operaciones de punto flotante debe ser manejada de manera cuidadosa, ya cualquier unidad de compilación con las opciones `-msse, -msse2, -mmmx -mavx` (instrucciones vectoriales y de aritmética de punto flotante) debe ser compilada de manera completamente aislada de otras unidades compiladas regularmente. Esto es así ya que, si estas instrucciones llegaran a alguna parte del código que no está envuelta por llamadas a `kernel_fpu_begin` y `kernel_fpu_end`, podrían ocurrir inconsistencias en los registros del tareas del *kernel*. La Figura 4.5 presenta cómo una función compilada con estas opciones debe estar siempre encapsulada por las funciónes `kenel_fpu_begin()` y `kernel_fpu_end()`. Mientras la llamada a un bloque de código que incluye operaciones de punto flotante se

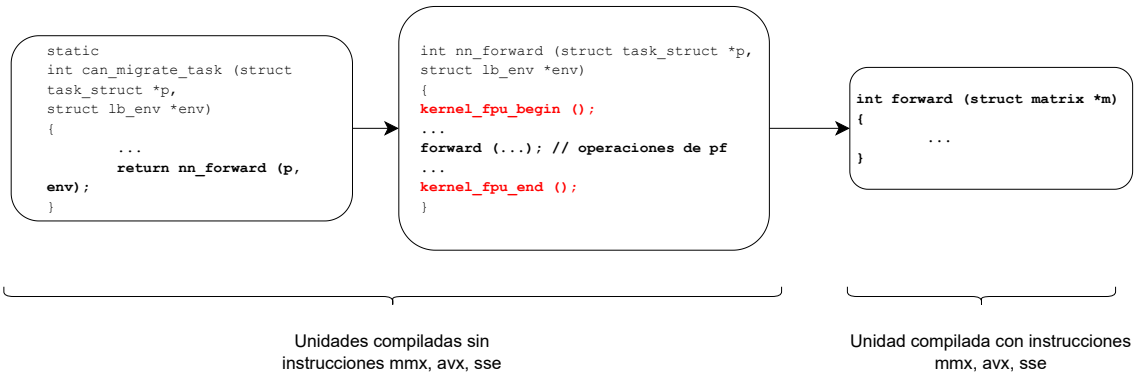


FIGURA 4.5: Representación gráfica de la manera correcta de compilar código que utiliza operaciones de punto flotante en Linux.

encuentre encapsulada por las rutinas mencionadas, estas operaciones no causarán problemas en el *kernel* ya que sus los estados de sus registros se manejarán correctamente.

Otra opción evaluada es utilizar aritmética de punto fijo para simular la aritmética de punto flotante que se realiza en la inferencia y entrenamiento del modelo. Para representar números decimales se utilizan variables enteras (usualmente de 32 *bits*) y se elige una cantidad de *bits* para representar su parte entera y el resto para representar su parte decimal. Esta técnica requiere una implementación más compleja y en la mayoría de los casos presenta un mayor margen de error a cambio de una mayor velocidad de procesamiento. En este trabajo, se opta por el uso de las rutinas `kernel_fpu_begin() y kernel_fpu_end()`.

4.1.6 Configuración del *Kernel*

Para integrar el modelo al *kernel*, se incluye la librería de funciones de inferencia como parte del código y se crea una configuración adicional en el menú de *Kconfig* que permite elegir al compilar si se usará el modelo para hacer inferencias sobre las migraciones agresivas, o si se optará por el funcionamiento tradicional para decidir el comportamiento del *load balancer*. La configuración agregada en el archivo `init/Kconfig` se

encuentra detallada en el Anexo 2. Se agregaron dos opciones de compilación: i) para elegir si utilizar o no el modelo de ML en la decisión de migración y ii) para elegir si compilar las operaciones de aritmética de punto flotante con las rutinas `kernel_fpu_...()` o simularlas con aritmética de punto fijo. Esta última se encuentra como opción pero actualmente no determina ningún cambio en el *kernel* dado que la lógica de aritmética de punto fijo no fue implementada para este trabajo.

Capítulo 5

RESULTADOS Y DISCUSIÓN

Este capítulo presenta los experimentos realizados con el modelo y se discuten los resultados obtenidos.

Con el objetivo de realizar pruebas preliminares del modelo propuesto, se realizaron tres experimentos: (i) replicar el modelo de [1] y entrenarlo con datos de diversas cargas de trabajo, (ii) modificar dicho modelo para predecir ocurrencias de migraciones forzadas en llamadas futuras con diversas cargas de trabajo y (iii) generar un modelo que logre predecir ocurrencias de migraciones forzadas en un instante de tiempo y probar su exactitud en espacio de *kernel* en tiempo real. El marco metodológico que se siguió para alcanzar estos objetivos consiste principalmente en cinco etapas: (i) la recolección de datos de llamadas a la función `can_migrate_task()` utilizando *kernel tracing*; (ii) el preprocesamiento de datos recolectados; (iii) el entrenamiento del modelo con los conjuntos de datos y las pruebas de precisión para los conjuntos de prueba. El alcance del tercer experimento incluye también (iv) la implementación del modelo en espacio de *kernel*, y (v) la evaluación del rendimiento del nuevo kernel modificado con diversas cargas de trabajo representadas por *benchmarks*, especificados en la Tabla 5.2. Los experimentos fueron ejecutados en un equipo personal, contando con las especificaciones mostradas en la Tabla 5.1. Los hiperparámetros y las dimensiones de las redes neuronales utilizadas para cada experimento se encuentran detalladas en la Tabla 5.8.

Componente	Especificación
CPU	Intel(R) Core(TM) i5-1035G1 CPU @ 1.00GHz
Cores	8
Nodos NUMA	1
Memoria principal (GB)	8

TABLA 5.1: Especificaciones de la máquina local utilizada para la recolección y entrenamiento de datos

Benchmark	Aplicación
PTS Cassandra	Escrituras y lecturas en un *cluster* Cassandra
PTS Arrayfire	Procesamiento numérico en CPU y GPU.
PTS Askap	Algoritmos de convolución y deconvolución de muestras del *Australian SKA Pathfinder*
PTS Compress-zstd	Compresión de una imagen de disco de *FreeBSD* utilizando la compresión zstd
Hackbench	Envió de mensajes entre una gran cantidad de procesos

TABLA 5.2: Lista de *benchmarks* utilizados para la recolección de datos y entrenamiento de modelos durante ambos experimentos, con su descripción correspondiente.

5.1 Experimento 1: Predicción de decisiones de CFS en un instante de tiempo

Se reprodujo el experimento realizado en el trabajo de [1] para definir un marco de trabajo sobre el cuál implementar posteriormente el modelo de predicción de migraciones forzadas.

5.1.1 Recolección de datos

A través de BCC se recolectaron los atributos listados en la Tabla 5.3 incluyendo a la mayoría de los utilizados para entrenar el modelo en el trabajo de [1], con la adición de algunos atributos correspondientes a la cola de ejecución de destino en el contexto de migración. Luego se separaron los datos en dos categorías: 1) registros que corresponden a una llamada a la función `can_migrate_task()` que resulta en la migración de una tarea de un CPU a otro y 2) registros que corresponden a una llamada a la función `can_migrate_task()` que no resultan en la migración de una tarea de un CPU a otro. Al hacer un análisis de correlación entre los atributos y sus categorías, se encontró que los atributos *task_util_avg, delta, task_nr_migrations, pid, sd_nr_balance_failed, task_vruntime, cpu_idle* son los que mejor representan a cada categoría ($n = 7$), por lo que se utilizaron únicamente estos datos para entrenar los modelos con el fin de reducir la dimensionalidad del modelo y consecuentemente el costo de las operaciones. Se observa que cada dataset correspondiente a un *benchmark* obtiene valores de correlación distintos para cada variable.

Estos atributos fueron determinados de la siguiente manera. Primero, se obtuvieron cinco atributos con mayor correlación para cada benchmark, y después se agruparon para conformar el vector de entrada del modelo. La Tabla 5.4 muestra los siete atributos resultantes de este proceso, siendo *delta* el de mayor *score* para este experimento, con valores más altos en los *benchmarks* `arrayfire` y `cassandra`. Este representa el tiempo de ejecución real de la tarea desde la última actualización de su `vruntime`. Un valor de score alto representa una alta correlación entre el atributo y la variable de salida, i.e. la ocurrencia de una migración forzada. Este método se utiliza para determinar los atributos utilizados en los experimentos restantes.

Atributo	Descripción
pid	Identificador (ID) del proceso
ts	Marca de tiempo del instante de recolección
task_load_avg	Carga de trabajo promedio de la tarea
task_util_avg	Utilización de CPU promedio de la tarea
task_vruntime	Valor actual de vruntime de la tarea
task_nr_migrations	Cantidad de veces que se ha migrado la tarea
start_runtime	Marca de tiempo de la última actualización del vruntime de la tarea
dst_rq_load	Carga de trabajo de la cola de ejecución de destino
dst_rq_load_avg	Carga de trabajo promedio de la cola de ejecución de destino
dst_rq_nr_running	Cantidad de tareas en la cola de ejecución de destino
dst_rq_util_avg	Utilización de CPU promedio de la cola de ejecución de destino
src_rq_load	Carga de trabajo de la cola de ejecución de origen
src_rq_load_avg	Carga de trabajo promedio de la cola de ejecución de origen
src_rq_nr_running	Cantidad de tareas en la cola de ejecución de origen
src_rq_util_avg	Utilización de CPU promedio de la cola de ejecución de origen
imbalance	Valor numérico de desbalance de cargas de trabajo entre las colas de un CPU domain
sd_cache_nice_tries	Límite de intentos fallidos de migración por política de cache nice
cpu_idle	Verdadero si el CPU se encuentra ejecutando la tarea idle
cpu_newly_idle	Verdadero si recientemente el CPU empezó a ejecutar la tarea idle
sd_nr_balance_failed	Cantidad de intentos de balance fallidos en el CPU domain
delta	Tiempo de ejecución real de la tarea desde la última actualización del valor de vruntime
migration	Decisión final de migración (0 cuando no hubo migración, 1 cuando hubo migración)

TABLA 5.3: Lista de atributos recolectados relacionados al contexto de la migración y a los procesos en ejecución y sus definiciones.

	Score				
Atributo	**arrayfire**	**askap**	**cassandra**	**compress-zstd**	**hackbench**
cpu_idle	7749.85	9691.54	732.98	7837.08	6656.93
pid	21.68	78.70	75.86	1212.13	405.16
task_util_avg	0.17	125.78	39.66	7010.51	455838.23
sd_nr_balance_failed	1298.48	632.54	39667.73	984.52	1.02
task_vruntime	11490.23	9453.88	351.49	7629.09	637.64
delta	192723.65	148431.65	162989.79	120962.92	90859.08
task_nr_migrations	226.87	1329.91	0.18	60.67	38141.56

TABLA 5.4: *Score* univariado de correlación con la salida, para cada variable utilizada en el experimento 1.

Benchmark	Cantida total de registros	Cantidad de migraciones forzadas	Cantidad de llamadas resultantes en una migración forzada	Porcentaje de migraciones forzadas (%)	Porcentaje de llamadas resultantes en una migración forzada (%)
PTS Cassandra	1000000	70197	257518	7.02	25.75
PTS Arrayfire	1000000	6531	44901	0.65	4.49
PTS Askap	1000000	3953	23195	0.40	2.32
PTS Compress zstd	1000000	6759	15845	0.67	1.58
Hackbench	1000000	29	63	0.003	0.006

TABLA 5.5: Porcentaje de migraciones forzadas, y el porcentaje de las llamadas que resultan en una eventual migración forzada (en el futuro)

5.1.2 Entrenamiento y validación del modelo

Se realizo la implementación del modelo al cuál llamamos NeuralCFS usando a) Pytorch y b) LibML. Para NeuralCFS_LibML se utilizaron arquitecturas con menos unidades (ver Tabla 5.8) ya que a pesar de que en este experimento no se hayan realizado pruebas en espacio de *kernel* esta librería implementa las funcionalidades para integrar los modelos de manera híbrida e idealmente estos deben ser compactos para no agregar *overhead* al *scheduler*. Esto se cumple para todos los experimentos que hacen uso de LibML. Los resultados de exactitud del entrenamiento de ambos modelos para cada *benchmark* se encuentran graficados en la Figura 5.1. Para la mayoría de *benchmarks* se alcanza una precisión superior al 90 %, a excepción de `compress_zstd` cuyos resultados se de exactitud fueron del 87.44 % con NeuralCFS_PyTorch y 85.9 % con NeuralCFS_LibML. El valor más alto fue obtenido tras entrenar ambos modelos con datos de ejecución de *hackbench*, mostrando un 99.38 % de exactitud con NeuralCFS_PyTorch y un 95.16 % con NeuralCFS_LibML. Esto se debe a que cada *benchmark* genera un tipo de carga diferente en el sistema, pudiendo alterar predictabilidad según los datos recolectados.

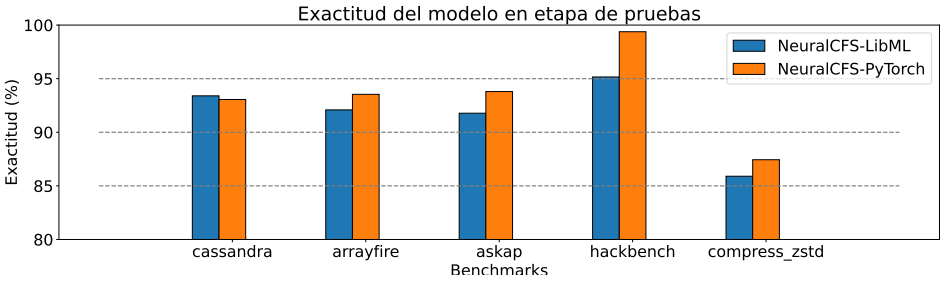


FIGURA 5.1: Gráfico de resultados de exactitud para predicciónes de decisiones de CFS en cada *benchmark*

5.2 Experimento 2: Predicción de migraciones agresivas en llamadas futuras

Una vez validados los modelos para la predicción de decisiones de CFS, se modificaron ligeramente con el fin de realizar predicciones sobre la ocurrencia de migraciones forzadas.

5.2.1 Recolección de datos

La recolección de datos reveló información importante sobre la ocurrencia de migraciones forzadas en las cargas de trabajo mencionadas. Como se observa en la Tabla 5.5, la ocurrencia de migraciones forzadas es más frecuente en `cassandra` con un 7.02 %, y más baja en `arrayfire` (0.65 %), `askap` (0.4 %)y `compress-zstd` (0.67 %). Hacemos dos observaciones principales. Primero, `hackbench` tuvo la menor frecuencia de ocurrencias con $0.01\,\%$ de las decisiones de un total de más de un millón de llamadas. Esto podría deberse a la cantidad de *threads* del *benchmark* en caso presenten contención por recursos de *cache*. Además, esto podría llevar a que una baja cantidad de threads se encuentren *cache-hot*. Segundo, se encontró que para estas categorías son diez las variables más representativas al realizar un análisis de correlación: *cpu_idle, task_nr_migrations, cpu_newly_idle, ts, delta, sd_cache_nice_tries, start_runtime, src_rq_nr_running, task_util_avg, sd_nr_balance_failed* ($n = 10$). Los *scores* de correlación para cada variable se encuentran en la Tabla 5.6.

Atributo	Score				
	arrayfire	**askap**	**cassandra**	**compress-zstd**	**hackbench**
cpu_idle	1497.37	1354.55	370.36	641.28	27.63
cpu_newly_idle	831.99	793.99	783.39	257.09	18.33
task_util_avg	1831.09	1873.33	16.74	904.71	10.00
sd_nr_balance_failed	3423.52	3.53	54.11	32.87	2.68
sd_cache_nice_tries	13.02	2251.77	16.90	910.06	2.00
delta	231.24	184.00	952.17	158.21	0.89
start_runtime	1461.39	1354.24	10472.13	0.19	0.33
task_nr_migrations	378.40	157.96	5913.19	89.62	0.27
ts	1741.94	1561.88	13004.84	8.87	0.21
src_rq_nr_running	88.71	35.70	58.87	20.31	0.17

TABLA 5.6: *Score* univariado de correlación con la salida, para cada variable utilizada en el experimento 2.

5.2.2 Entrenamiento y validación del modelo

El entrenamiento se realizó de la misma manera que el experimento anterior, con ligeros cambios en la dimensionalidad por la cantidad de variables con alta correlación (en este caso 10). Los resultados del entrenamiento muestran que la exactitud para los *benchmarks* `cassandra`, `arrayfire`, `askap`, `hackbench` excede el 90 % con ambas herramientas utilizadas, siendo `hackbench` el caso para el cuál se alcanzó la mayor exactitud con un 100 % para NeuralCFS_PyTorch y 96.8 % para NeuralCFS_LibML. Además del entrenamiento y validación, se realizaron pruebas con el modelo utilizando un conjunto de datos completamente separados, proveniente de otra ejecución del mismo *kernel*. Los rsultados para los tres escenarios se encuentran graficados en la Figura 5.2, donde hallamos dos observaciones principales. Primero, al igual que el experimento anterior, el *benchmark* `compress-zstd` tuvo el menor valor de exactitud, con un 83.8 % para NeuralCFS_PyTorch y 80.6 % para NeuralCFS_LibML en la etapa de entrenamiento. Segundo, los resultados de estas pruebas comprueban la efectividad del modelo y descartan la posibilidad de un sobreajuste, al menos dentro del mismo *kernel*, ya que todas obtuvieron valores de exactitud similares o incluso superiores al de la etapa de entrenamiento. Esto a excepción del caso de *hackbench*, ya que este tuvo una exactitud

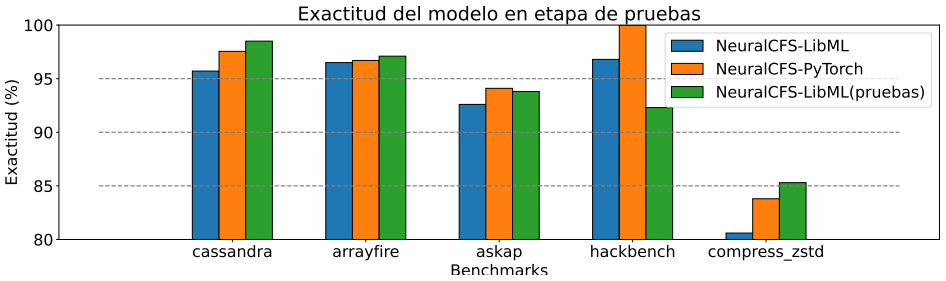


FIGURA 5.2: Resultados de exactitud de la predicción de migraciones forzadas en llamadas futuras para cada *benchmark*

considerablemente inferior del 92.3 % posiblemente debido a la poca cantidad de datos registrados de migraciones forzadas.

5.3 Experimento 3: Predicción de migraciones agresivas en un instante de tiempo

Finalmente, se utilizaron los mismos modelos para predecir esta vez la ocurrencia de migraciones forzadas en un instante de tiempo, es decir, en una misma llamada a la función `can_migrate_task()`.

5.3.1 Recolección de datos

Se utilizaron los mismos datos recolectados para los experimentos anteriores, y se definió nuevamente las dos categorías: 1) registros de una llamada a `can_migrate_task()` que resultan inmediatamente en una migración forzada y 2) registros de una llamada a `can_migrate_task()` que no resultan inmediatamente en una migración forzada.

A diferencia del método de preprocesamiento utilizado en el experimento 2, en este conjunto de datos se clasificaron los datos basándose únicamente en el valor de retorno

Atributo	Score				
	arrayfire	**askap**	**cassandra**	**compress-zstd**	**hackbench**
cpu_idle	20,935.82804900	2,788.98044100	3.15614000	13,673.92620400	0.19064700
cpu_newly_idle	3,529.30756300	1,270.87435300	8,365.95049700	2,402.05320700	0.31026200
task_util_avg	13,313.01718400	5,138.66228900	330.94111500	4,487.72621800	0.00498620
sd_nr_balance_failed	55,256.59555200	37,948.11465200	108,486.86007600	49,143.48607400	3,252.76558300
sd_cache_nice_tries	2,259.61971700	665.73516600	1,529.52103300	247.79226600	17.77254700
delta	293.72048300	154.35981900	2,645.16339200	211.99061600	15.42653700
dst_cpu	0.73195900	0.20086400	395.36024300	199.45679400	0.10931100

TABLA 5.7: *Score* univariado de correlación con la salida, para cada variable utilizada en el experimento 3.

de la función `can_migrate_task()`. Se encontró que las variables con mayor correlación a las categorías fueron: *cpu_idle, cpu_newly_idle, task_util_avg, sd_nr_balance_failed, sd_cache_nice_tries, delta* ($n = 7$).

5.3.2 Entrenamiento y validación del modelo

El entrenamiento y la validación del modelo fueron realizados de la misma manera que en los experimentos anteriores. Los resultados en promedio fueron de 97.6 % con NeuralCFS_PyTorch y de 97.4 % con NeuralCFS_LibML, como se observa en la Figura 5.3. Además, en este gráfico se observa un tercer valor correspondiente a la exactitud de las inferencias del modelo entrenado con LibML posteriormente integrado al *scheduler* en tiempo real. Aquí hay dos principales observaciones. Primero, el modelo que alcanzó mayor exactitud fue aquel entrenado con el conjunto de datos de `hackbench`, con un 100 % de precisión en la etapa de entrenamiento (utilizando ambas herramientas), aproximadamente 4.5 % mayor al del modelo con más baja exactitud entrenado con datos de `cassandra` (95.76 % con PyTorch y 95.44 % con LibML). Segundo, podemos ver que para la mayoría de los casos la exactitud de las predicciones disminuye al ser realizadas en tiempo real. Para la mayoría de *benchmarks* esto no representa una pérdida de más del 1.5 % de exactitud, con la única excepción siendo el caso de `hackbench` donde la precisión cae de 100 % a 96.83 %, debido al sobreajuste ocasionado por la escasez de datos

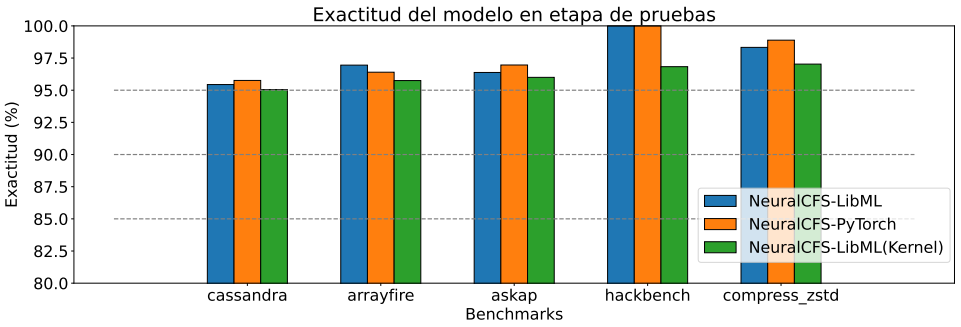


FIGURA 5.3: Resultados de exactitud para la predicción de migraciones forzadas en un instante de tiempo

de migraciones forzadas recolectados durante su ejecución.

Para integrar el modelo en espacio de *kernel*, se utilizó la *system call* descrita en la sección 4.1.2. Para cada experimento, se repitió la etapa de entrenamiento utilizando LibML con el paso adicional de, una vez entrenado el modelo, llamar a esta *system call* para enviar la información del modelo a un componente de *kernel* implementado para leerla, y posteriormente construir y utilizar el modelo para inferencias en tiempo real. Al entrenar y enviar el modelo a *kernel space*, este automáticamente empieza a ser utilizado, por lo que los *benchmarks* fueron ejecutados justo después de que este proceso se completara. Esto se repitió para cada *benchmark*.

5.3.3 Análisis de *performance* en espacio de *kernel*

Dado que para este experimento se utilizó el modelo para hacer inferencias en espacio de *kernel* diréctamente, se realizó un análisis del costo a nivel de *performance* que trae consigo esta integración. Además, se incluyó el modelo utilizado en el experimento 5.2 en esta experimentación y se le permitió tomar las decisiones del *scheduler* para analizar su impacto. Para ello, se recolectaron datos de dos maneras: 1) se obtuvieron datos de latencia de la función `can_migrate_task()` a través del mismo método para la

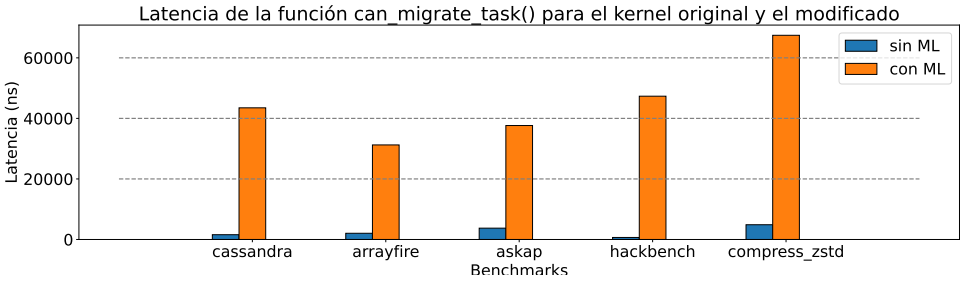


FIGURA 5.4: Resultados de latencia de la función `can_migrate_task()` para ambos *kernel*

recolección de datos de los experimentos (utilizando BCC) con la diferencia de que únicamente se recolectaron datos de marcas de tiempo al entrar y al salir de la función, y 2) se obtuvieron datos del tiempo de ejecución de cada *benchmark* a través de la información brindada por los *suites* de pruebas de los mismos. Los resultados de latencia para la función `can_migrate_task()` se encuentran graficados en la Figura 5.4, donde se muestran estos valores para cada *benchmark* al ser ejecutados en ambos *kernel*, y listados en la Tabla 5.9. En la Figura 5.6 se observa la función de densidad de probabilidad de latencias promedio al ejecutar cada *benchmark* de ambos *kernel* y en la Figura 5.5 se observan los promedios del tiempo de ejecución de cada *benchmark* al utilizar ambos *kernel*. Además, los valores de estos resultados están listados en la Tabla 5.10. Aquí se hacen dos observaciones principales. Primero, el uso de inferencias con el modelo en espacio de *kernel* trae consigo una latencia promedio de `can_migrate_task()` aproximadamente $\times 27$ mayor a la del *kernel* original. Sin embargo, esto parece no tener un fuerte impacto sobre el tiempo de ejecución de los *benchmarks* a nivel general, ya que los tiempos de ejecución para el *kernel* con ML son 2 % menores que los del *kernel* original en promedio. Segundo, La predicción y toma de decisiones del modelo generado en el experimento 5.2 tampoco tuvo un impacto significativo en el tiempo de ejecución, teniendo un tiempo de ejecuión promedio 2.3 % menor al del *kernel* original.

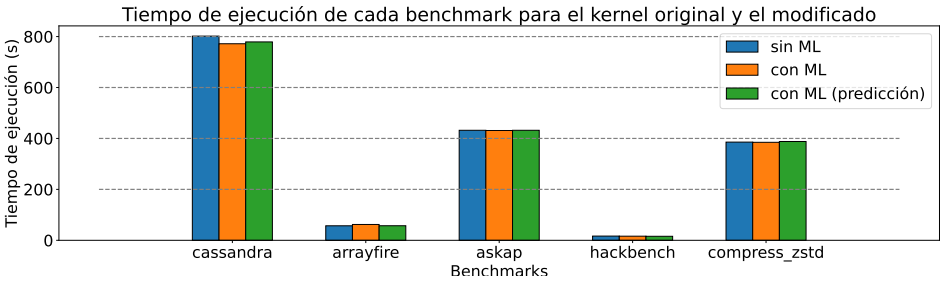


FIGURA 5.5: Resultados de tiempo de ejecución de los *benchmarks* para ambos *kernel*

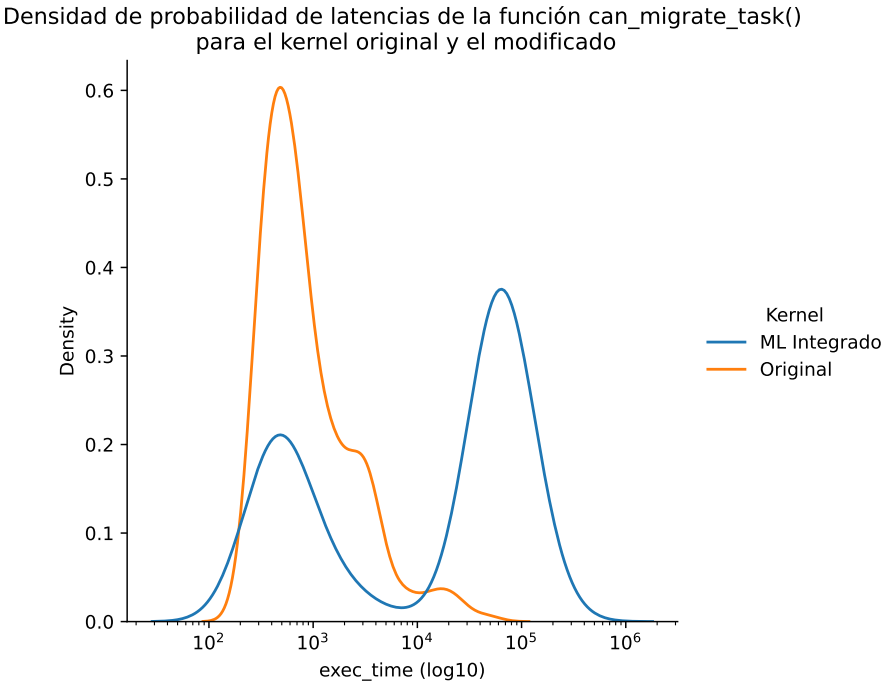


FIGURA 5.6: Función de densidad de probabilidad para la latencia de `can_migrate_task()` de ambos *kernel* al ejecutar cada *benchmark*.

	Experimento 1		Experimento 2		Experimento 3	
	NeuralCFS-PyTorch	**NeuralCFS-LibML**	**NeuralCFS-PyTorch**	**NeuralCFS-LibML**	**NeuralCFS-PyTorch**	**NeuralCFS-LibML**
N. de entradas (n)	7	7	10	10	7	7
N. de capas internas (l)	1	1	1	1	1	1
N. total de unidades	51	17	72	20	51	17
Tasa de aprendizaje	0.001	0.1	0.001	0.1	0.001	0.1
Decadencia de pesos	0.00001	-	0.00001	-	0.00001	-
Tamaño de batches	32	1	32	1	32	1

TABLA 5.8: Hiperparámetros utilizados para las redes en cada experimento. La decadencia de pesos fue utilizada sólo en PyTorch.

	Benchmarks				
	Hackbench	**Cassandra**	**Compress-ZSTD**	**Arrayfire**	**Askap**
Kernel Original	669.02	1578.89	4871.05	2051.96	3743.49
Kernel con ML	47335.62	43490.65	67454.05	31235.13	37644.60

TABLA 5.9: Latencias promedio (ns) de la función `can_migrate_task()` del *kernel* original y el modificado durante la ejecución de cada *benchmark*.

	Hackbench	**Cassandra**	**Compress-ZSTD**	**Arrayfire**	**Askap**
Kernel Original	16.59	801.8	385.6	57	432
Kernel con ML	16.19	771.8	384.8	62.2	431.2
Kernel con ML (predicción)	15.78	779	388	57.2	432

TABLA 5.10: Tiempo de ejecución (s) promedio de cada *benchmark*.

CONCLUSIONES

Este trabajo propuso el uso de un modelo adaptable de ML basado en el de [1] para predecir la ocurrencia de de migraciones forzadas en el *scheduler* CFS de *Linux*. Se logró generar y entrenar modelos con datos de llamadas a la función `can_migrate_task` recolectados través de BCC durante la ejecución de los *benchmarks* `hackbench`, `arrayfire`, `askap`, `cassandra` y `compress-zstd`. Se extendió el trabajo en [1] entrenando un modelo para la predicción de decisiones de CFS (si una tarea es migrada o no) con los datos de cada *benchmark*. Los resultados de exactitud no fueron tan altos como los de su trabajo para todos los modelos, sin embargo cabe resaltar que los experimentos no fueron realizados en un sistema que comparta las características de aquel utilizado en su experimentación, y también que existe una correlación entre la naturaleza de una carga de trabajo y la predictabilidad de las decisiones de CFS, que afecta la capacidad del modelo.

En base a los análisis realizados sobre los datos recolectados, concluimos que cada carga de trabajo lleva al sistema a estados diferentes, resultando en diferentes correlaciones entre ciertas variables del contexto de ejecución y las decisiones de migración. En otras palabras, las variables más significativas pueden variar entre diferentes cargas de trabajo. Esto sugiere dos cosas: 1) que la dimensionalidad de los vectores de entrada puede reducirse drásticamente dependiendo de la cantidad de variables con alta correlación

con la salida, y 2) que la capacidad de adaptar el modelo a cada carga de trabajo trae mejoras en relación a la exactitud de las predicciones, como se observa en casos como el del *benchmark* `cassandra`, que obtuvo resultados de hasta 98.5 % de exactitud mientras que `compress-zstd` que alcanzo hasta un 85.3 % de exactitud al predecir migraciones en un instante futuro.

Es importante resaltar que en la mayoría de escenarios, la ocurrencia de migraciones forzadas y de llamadas resultantes en migraciones forzadas representan un porcentaje muy bajo de las llamadas totales, reflejando las limitaciones del potencial beneficio del uso de los modelos propuestos. En el mejor de los casos (para la ejecución de `PTS cassandra`), un modelo óptimo podría predecir y migrar prematuramente el 25 % del total de las llamadas.

La predicción de migraciones forzadas a través de redes neuronales resultó posible con una alta exactitud durante la ejecución de las cinco diferentes cargas de trabajo. El modelo predictor de migraciones forzadas en un instante de tiempo (dentro de la misma llamada) tuvo una exactitud de aproximadamente 97.42 % en las pruebas en espacio de usuario y de 96.13 % en pruebas en espacio de *kernel* en promedio para todos los *benchmarks*. Por otro lado, el modelo predictor de migraciones forzadas en llamadas futuras tuvo una exactitud de 92.442 % en pruebas en espacio de usuario. Este modelo se integró en espacio de *kernel* para tomar decisiones de migraciones y medir el impacto de esta integración en el tiempo de ejecución de los *benchmarks*. Como se observa en los resultados, esta metodología no tiene un impacto significativo en el *performance* del *kernel*, sin embargo refuerzan la idea de que la integración de ML es diréctamente aplicable en espacio de *kernel* y factible sin generar pérdidas significativas.

TRABAJO FUTURO

El desarrollo de este sistema abre las puertas a una gran cantidad de posibles experimentaciones, ya sea utilizando redes neuronales en diferentes partes del *kernel*, o utilizando otras técnicas de ML y comparando su efectividad en contraste con lo obtenido en este trabajo. Esto puede ser interesante ya que permite realizar mediciones sobre el consumo adicional de recursos que demanda el uso de modelos de ML, probar distintas configuraciones y explorar más opciones para la tediosa tarea de integrarlos en el *kernel*. Por otro lado, la experimentación fue realizada sin una manera robusta de analizar el uso de memoria por parte del modelo, que es una métrica importante que analizar al hablar del costo del uso del modelo. Trabajos futuros en curso involucran la ejecución del *kernel* modificado en *hardware* dedicado, el análisis exhaustivo de *performance* y otras métricas relevantes como uso de memoria, y la implementación de la metodología presentada utilizando aritmética de punto fijo.

ANEXOS

```
config ML_FORCE_MIGRATIONS
    bool "ML prediction of forced migrations in CFS"
    default y
    help
      Select 'y' to activate the prediction of forced migrations in the CFS
      scheduler using a neural network.

config FIXED_POINT_ML
    bool "Fixed point arithmetic for ML operations"
    default n
    depends on ML_FORCE_MIGRATIONS
    help
      Use fixed-point arithmetics for inference operations.
```

LISTING 2: Opciones agregadas al menú de configuración del *kernel* en el archivo `init/Kconfig`

REFERENCIAS BIBLIOGRÁFICAS

[1] J. Chen, S. S. Banerjee, Z. T. Kalbarczyk, and R. K. Iyer, "Machine learning for load balancing in the Linux kernel," *APSys 2020 - Proc. 2020 ACM SIGOPS Asia-Pacific Work. Syst.*, no. Ml, pp. 67–74, 2020.

[2] G. Blake, R. G. Dreslinski, and T. Mudge, "A survey of multicore processors: A review of their common attributes," *IEEE Signal Process. Mag.*, vol. 26, no. 6, pp. 26–37, 2009.

[3] J. Kwack, T. Applencourt, C. Bertoni, Y. Ghadar, H. Zheng, C. Knight, and S. Parker, "Roofline-based performance efficiency of hpc benchmarks and applications on current generation of processor architectures," in *2019 Cray User Group Meeting*, vol. 5, 2019.

[4] D. Stanzione, J. West, R. T. Evans, T. Minyard, O. Ghattas, and D. K. Panda, "Frontera: The evolution of leadership computing at the national science foundation," in *Practice and Experience in Advanced Research Computing*, 2020, pp. 106–111.

[5] X. Iturbe, B. Venu, E. Ozer, J.-L. Poupat, G. Gimenez, and H.-U. Zurek, "The arm triple core lock-step (tcls) processor," *ACM Transactions on Computer Systems (TOCS)*, vol. 36, no. 3, pp. 1–30, 2019.

[6] P. J. Denning and T. G. Lewis, "Exponential laws of computing growth," *Commun. ACM*, vol. 60, no. 1, pp. 54–65, 2017.

[7] I. Kang, “The art of scaling: Distributed and connected to sustain the golden age of computation,” in *2022 IEEE International Solid- State Circuits Conference (ISSCC)*, vol. 65, 2022, pp. 25–31.

[8] S. Zhuravlev, J. C. Saez, S. Blagodurov, A. Fedorova, and M. Prieto, “Survey of scheduling techniques for addressing shared resources in multicore processors,” *ACM Comput. Surv.*, vol. 45, no. 1, pp. 1–28, 2012.

[9] S. Boyd-Wickizer, R. T. Morris, M. F. Kaashoek *et al.*, “Reinventing scheduling for multicore systems.” in *HotOS*, 2009.

[10] S. Kundan and I. Anagnostopoulos, “Priority-aware scheduling under shared-resource contention on chip multicore processors,” in *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2021, pp. 1–5.

[11] S. Zhuravlev, S. Blagodurov, and A. Fedorova, “Addressing shared resource contention in multicore processors via scheduling,” *ACM Sigplan Notices*, vol. 45, no. 3, pp. 129–142, 2010.

[12] R. Das, R. Ausavarungnirun, O. Mutlu, A. Kumar, and M. Azimi, “Application-to-core mapping policies to reduce memory system interference in multi-core systems,” in *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2013, pp. 107–118.

[13] T. Cucinotta, G. Lipari, and L. Schubert, *Operating System and Scheduling for Future Multicore and Many-Core Platforms*, 2017, no. January.

[14] S. Baruah and J. Carpenter, “Multiprocessor fixed-priority scheduling with restricted interprocessor migrations,” in *15th Euromicro Conference on Real-Time Systems, 2003. Proceedings.*, 2003, pp. 195–202.

[15] “The linux kernel archives,” accesed on 11/01/2024. [Online]. Available: https://www.kernel.org/

[16] J. Han and S. Lee, "Performance Improvement of Linux CPU Scheduler Using Policy Gradient Reinforcement Learning for Android Smartphones," *IEEE Access*, vol. 8, pp. 11 031–11 045, 2020.

[17] J. Lawall, H. Chhaya-Shailesh, J.-P. Lozi, B. Lepers, W. Zwaenepoel, and G. Muller, "Os scheduling with nest," *Proceedings of the Seventeenth European Conference on Computer Systems*, 2022.

[18] T. Helmy, S. Al-Azani, and O. Bin-Obaidellah, "A machine learning-based approach to estimate the cpu-burst time for processes in the computational grids," *2015 3rd International Conference on Artificial Intelligence, Modelling and Simulation (AIMS)*, 2015.

[19] A. Negi and P. Kumar, "Applying machine learning techniques to improve linux process scheduling," *TENCON 2005 - 2005 IEEE Region 10 Conference*, 2005.

[20] P. Ojha, S. R. Thota, V. M, and M. P. Tahilianni, "Learning scheduler parameters for adaptive preemption," *Computer Science Information Technology (CS IT)*, 2015.

[21] V. M. Safarzadeh and H. G. Loghmani, "Artificial intelligence in the low-level realm - A survey," *CoRR*, vol. abs/2111.00881, 2021. [Online]. Available: https://arxiv.org/abs/2111.00881

[22] J. Roberson, "{ULE}: A modern scheduler for {FreeBSD}," in *BSDCon 2003 (BSDCon 2003)*, 2003.

[23] Kernel.org git repositories, "Linux kernel source tree," accesed on 11/01/2024. [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git

[24] J. A. Brown, L. Porter, and D. M. Tullsen, "Fast thread migration via cache working set prediction," in *2011 IEEE 17th International Symposium on High Performance Computer Architecture*. IEEE, 2011, pp. 193–204.

[25] I. U. Akgun, A. S. Aydin, A. Shaikh, L. Velikov, and E. Zadok, "A machine learning framework to improve storage system performance," *Proceedings of the 13th ACM Workshop on Hot Topics in Storage and File Systems*, 2021.

[26] J. Zhang, C. Zeng, H. Zhang, S. Hu, and K. Chen, "Liteflow," *Proceedings of the ACM SIGCOMM 2022 Conference*, 2022.

[27] N. Jay, N. Rotman, B. Godfrey, M. Schapira, and A. Tamar, "A deep reinforcement learning perspective on internet congestion control," in *Proceedings of the 36th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, K. Chaudhuri and R. Salakhutdinov, Eds., vol. 97. PMLR, 09–15 Jun 2019, pp. 3050–3059. [Online]. Available: https://proceedings.mlr.press/v97/jay19a.html

[28] Y. Ma, H. Tian, X. Liao, J. Zhang, W. Wang, K. Chen, and X. Jin, "Multi-objective congestion control," in *Proceedings of the Seventeenth European Conference on Computer Systems*, ser. EuroSys '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 218–235. [Online]. Available: https://doi.org/10.1145/3492321.3519593

[29] S. Ha, I. Rhee, and L. Xu, "Cubic: A new tcp-friendly high-speed tcp variant," *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 5, p. 64–74, jul 2008. [Online]. Available: https://doi.org/10.1145/1400097.1400105

[30] Y. Qiu, H. Liu, T. Anderson, Y. Lin, and A. Chen, "Toward reconfigurable kernel datapaths with learned optimizations," *Proceedings of the Workshop on Hot Topics in Operating Systems*, 2021.

[31] H. A. Maruf and M. Chowdhury, "Effectively prefetching remote memory with leap," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, Jul. 2020, pp. 843–857. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/al-maruf

[32] IEEE, "Ieee standard for information technology—portable operating system interface (posix®) base specifications, issue 7," IEEE, Tech. Rep., Year, standard IEEE Std 1003.1-2017.

[33] TOP500: The List, "List statistics," accessed on 11/01/2024. [Online]. Available: https://www.top500.org/statistics/list/

[34] IDC: International Data Corporation, "Smartphone market share," accesed on 11/01/2024. [Online]. Available: https://www.idc.com/promo/smartphone-market-share

[35] M. Gebai and M. R. Dagenais, "Survey and analysis of kernel and userspace tracers on linux," *ACM Computing Surveys*, vol. 51, no. 2, p. 1–33, 2018.

[36] eBPF, "What is ebpf?" accessed on 11/01/2024. [Online]. Available: https://ebpf.io/what-is-ebpf/

[37] R. Love, *Linux kernel development*. Addison-Wesley, 2015, ch. No (Easy) Use of Floating Point, p. 20.

[38] L. Torvalds, "Kernel floating-point," accessed on 11/01/2024. [Online]. Available: https://yarchive.net/comp/linux/kernel_fp.html

[39] C. M. Bishop, *Pattern recognition and machine learning*. Springer, 2016, ch. Neural Networks, pp. 225–249.

[40] A. Qayyum, J. Qadir, M. Bilal, and A. Al-Fuqaha, "Secure and robust machine learning for healthcare: A survey," *IEEE Reviews in Biomedical Engineering*, vol. 14, pp. 156–180, 2021.

[41] F. Schroff, D. Kalenichenko, and J. Philbin, "Facenet: A unified embedding for face recognition and clustering," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015.