Facultad
de
Ciencias

Implementación de Algoritmo Backfilling para Planificación de Trabajos en Datacenters

(Backfilling Algorithm Implementation for Job Scheduling in Datacenters)

Trabajo de Fin de Grado
para acceder al

GRADO EN INGENIERÍA INFORMÁTICA

Autor: Javier Canales García

Director: José Luis Bosque Orero

Codirector: Mario Ibáñez Bolado

Julio-2022

Índice general

Índice de figuras

Índice de cuadros

Agradecimientos

En primer lugar, a mi familia, que siempre ha estado ahí para apoyarme independientemente de la situación y que gracias a su trabajo, esfuerzo y cariño me han ayudado a llegar donde estoy tanto académicamente como en la vida.

A mis amigos, con los que he pasado la mayor parte del tiempo estos últimos cuatro años y que también me han mostrado su apoyo. Juntos hemos pasado buenos momentos, aunque también hemos tenido tiempo para desahogarnos cuando las cosas no salían como queríamos.

A los profesores que he tenido durante toda mi vida académica, que me transmitieron su conocimiento e hicieron de mí una mente inquieta que busca aprender cosas nuevas cada día.

Y por último a José Luis y a Mario, que se interesaron en que desarrolláramos juntos este trabajo y me han guiado semana a semana para avanzar y finalizar satisfactoriamente este proyecto.

Resumen

El auge de nuevas tecnologías y las continuas mejoras en el ámbito de la informática han permitido el desarrollo de los conocidos como centros de datos o datacenters, que permiten llevar a cabo computación de altas prestaciones, así como otros muchos servicios en la nube. Este tipo de entornos no está pensado para ejecutar las tareas que un usuario habitual puede necesitar, si no que son requeridos para el tratamiento masivo de datos, aplicaciones científicas, medicina, etc. Los datacenters, usualmente, están formados por recursos computacionales heterogéneos para dar cabida a todo tipo diferente de tareas.

Las tareas que envían los usuarios a la plataforma, deben ser asignadas en los diferentes recursos computacionales disponibles de la forma más óptima posible para así, maximizar el uso y rendimiento de la infraestructura. Esta es un área muy compleja, en la que hay un gran número de factores a tener en cuenta, por lo que no existe una solución que sea válida y adecuada para todas las situaciones. Los datacenters deben disponer de un software con un cierto grado de inteligencia para ser capaces de decidir cuál es la correspondencia más adecuada entre tareas y recursos, denominado *planificador*.

Lo ideal en este tipo de entornos es disponer de un amplio abanico de políticas de planificación para seleccionar una u otra en función de la situación. Dependiendo de los parámetros de entrada que se reciban y los objetivos de planificación que se indiquen, ser capaces de adaptar el entorno computacional para satisfacer la demanda.

Por todo esto, el propósito de este proyecto es analizar un algoritmo de planificación que es altamente utilizado hoy en día en los datacenters reales, conocido como *Backfilling*, y realizar su implementación en el simulador de datacenters heterogéneos IRMaSim. Adicionalmente se realiza un estudio de su comportamiento frente a otros algoritmos de planificación para ver sus virtudes y defectos.

Palabras claves: Simulador de datacenters heterogéneos, Backfill, Planificación en datacenters, IRMaSim, Algoritmo de planificación

Abstract

The rise of new technologies and the continuous improvement in the ambit of computering has allowed the development of the commonly known as datacenters, that allow to carry out high performance computing. This type of environment is not suitable to execute tasks that a regular user will need to do, but they are needed carry out the treatment of massive data, scientific applications, medicine, etc. Datacenters, usually, are build with heterogeneous computing resources so they can manage a lot of different type of tasks.

The tasks that users sent to the platform, should be mapped on the different computng resources available in the most optimum way so the use and performance of the infraestructure can be maximized. This is a very complex field, where there are a huge amount of facts to take into account, so there is no a solution which is valid and suitable in all the situations. Datacenters must have a software with a certain level of intelligence to be able to choose which is the perfect match between tasks and resources, known as *scheduler*.

The ideal on this type of environments would be to have a bunch of scheduling policies so it can choose one or another depending on the situation. Depending on the input parameters and the scheduling objectives that the user provides, the environment should be able to change its behavior and satisfy the demand.

Taking all this into account, the main purpose of this project is analyze a scheduling algorithm that is highly usead on real datacenters nowadays, known as *Backfilling*, and make its implementation on the heterogeneous datacenter simulator IRMaSim. Besides we carry out a study of its behavior comparing it with other scheduling algorithms to check its strengths and weaknesses.

Keywords: Heterogeneous datacenter simulator, Backfill, Datacenter scheduling, IRMaSim, Scheduling algorithm

Capítulo 1

Introducción

En este capítulo se describirán las líneas base sobre las que se realiza el desarrollo del proyecto. Se presentará el problema que supone la planificación de tareas en los centros de computación de alto rendimiento, y por qué no existe una solución que sea única y óptima para todos los casos. Todas las nuevas propuestas para mejoras en datacenters se llevan a cabo en simuladores ya que no se dispone de un centro operativo completo para pruebas. Además, se detallarán los objetivos fundamentales a conseguir con este trabajo, junto con el plan de trabajo seguido para su realización y la estructura utilizada en el documento.

1.1. Planificación de tareas en datacenters

Los ordenadores de propósito general disponen de potencia suficiente para las tareas cotidianas que realizan la gran mayoría de personas, pero cuando hablamos de computación de alto rendimiento estos se quedan cortos, y es necesaria una combinación de diversos componentes, tanto hardware como software, para alcanzar ciertas cotas [2]. Gracias a este rendimiento se pueden realizar tareas a priori costosas con una cierta rapidez y eficiencia que, realizadas con computadores convencionales sería mucho más demandante. Este tipo de computación está ligado al uso de los conocidos como datacenters, los cuáles disponen de la infraestructura necesaria para obtener rendimientos tan altos [3].

Estos datacenters están compuestos por miles de recursos computacionales que, por lo general presentan una arquitectura de carácter heterogéneo [4]. Esto significa que los componentes con los que nos encontramos son de diferente naturaleza, cada uno con sus propias características y arquitectura, y son empleados en conjunto para satisfacer las tareas que llegan al centro de datos [5]. Cuando estas llegan, son almacenadas y quedan a la espera de ser ejecutadas en el momento que el datacenter disponga de los recursos necesarios.

Adicionalmente a la heterogeneidad, otro problema que afecta a la complejidad del data-

center, son los objetivos de planificación. Estos son empleados para definir cómo queremos que ejecute el conjunto de tareas enviado al sistema. El usuario puede optar por tratar de optimizar diversos parámetros como pueden ser el rendimiento o la eficiencia energética. Este último en especial, está adquiriendo una elevada importancia en los últimos años, ya que los costes asociados están en aumento y el impacto en el medio ambiente que puede generar este tipo de estructuras puede llegar a ser excesivamente alto [6].

La planificación de tareas en datacenters, como se verá más adelante, resulta ser un problema de decisión NP-completo. Esto significa que para cada situación planteada, no existe una solución que sea única y óptima, si no que en función de como sean los parámetros recibidos se debe optar por una solución u otra [7]. Para resolver este tipo de problemas se tiende a emplear algoritmos heurísticos debido a que los algoritmos tradicionales no son capaces de dar una solución exacta, o consiguen alcanzar la solución pero en un tiempo que no es asumible [8]. Debido a esto la inteligencia artificial y las técnicas de aprendizaje reforzado son muy apropiadas para la planificación de datacenters, ya que, en base a las ejecuciones pasadas son capaces de predecir cuál es la forma óptima de tratar un conjunto de tareas dado [9, 10].

Debido a esta complejidad, la búsqueda de puntos de mejora en el área de los datacenters resulta puntero actualmente y por ello, grupos de investigación de todo el mundo están trabajando para encontrar nuevas maneras de aumentar el rendimiento. Estas nuevas mejoras ya no son solo en la planificación de tareas, si no también en el desarrollo de nuevos componentes que superen en capacidad a los actuales o nuevas formas de interconectar todos los elementos para que las velocidades de transferencia de datos aumenten y por tanto la latencia disminuya.

Las constantes mejoras propuestas, antes de hacerse efectivas, deben contrastarse en base a experimentación. No se puede optar por poner en marcha una nueva solución en un entorno real sin antes estar completamente seguros de su correcto funcionamiento. Esto podría resultar fatal ya que en los peores casos, el datacenter al completo quedaría inutilizado, con las consecuentes pérdidas en términos de costes. Lo habitual no es tener el datacenter replicado al completo para poder verificar que las nuevas implementaciones son funcionales en la realidad, debido a aspectos de espacio, coste y consumo energético. Por esto, los simuladores son muy apropiados para este tipo de desarrollos, ya que consiguen representar en gran medida la realidad, obteniendo unos resultados con una cota de error muy baja y unos speedups en cuanto a rendimiento bastante altos en comparación con la ejecución en un cluster real.

En este ámbito nace IRMaSim, una herramienta de simulación desarrollada en el seno del grupo de investigación de Arquitectura y Tecnología de Computadores de la Universidad de Cantabria, y del que ya se han realizado algunas publicaciones [11].Es un simulador de datacenters heterogéneos que permite representar los diferentes componentes con un nivel de detalle acorde al problema que se quiere tratar, y realizar ejecuciones de planificación de tareas con diversos algoritmos. La herramienta permite conocer detalles como la latencia, el consumo energético, el uso de la memoria, o la traza resultante de la ejecución de los jobs con diferentes parámetros por cada uno de ellos.

Este Trabajo Fin de Grado continua con el desarrollo de IRMaSim, centrándose en los objetivos específicos que se describen en la siguiente sección.

1.2. Objetivos

El propósito fundamental de este proyecto es agregar un nuevo tipo de planificador, conocido como *Backfill*, al simulador IRMaSim. Este tipo de planificador es muy utilizado actualmente en herramientas como *SLURM* puesto que permite aprovechar mejor las capacidades del datacenter mientras que otros tipos de algoritmos bloquearían su ejecución. El trabajo se desglosa principalmente en dos tareas fundamentales:

- *Adición de un nuevo algoritmo de planificación en el simulador*: Las posibilidades de elección en cuanto a la política de planificación en la herramienta IRMaSim, por el momento son algo limitadas ya que únicamente se dispone de dos tipos de planificadores. Se pretende añadir un nuevo algoritmo para dotar de un mayor realismo y flexibilidad al simulador, y que el usuario sea capaz de poder analizar cuál es el que más le conviene en cada caso; además de ajustar el diseño del simulador para que una futura implementación de sucesivos algoritmos se realice de una forma más sencilla.
- *Comparativa con algoritmos clásicos*: Una vez realizada la implementación en la herramienta, se propone realizar un estudio acerca de diversas métricas como pueden ser rendimiento, eficiencia energética, etc; para así poder, en primer lugar asegurar el correcto funcionamiento del nuevo algoritmo y que este se ciña a un comportamiento real, y a continuación realizar una comparativa de sus beneficios y desventajas frente a los demás algoritmos disponibles en la herramienta, y la ejecución en un cluster real.

1.3. Plan de trabajo

Para la consecución de los objetivos descritos en la Sección 1.2 se ha seguido el siguiente plan de trabajo:

- *Estudio de la planificación en datacenters*: En primer lugar resulta fundamental conocer cómo se producen los procesos de planificación en un datacenter. Esto permite conocer en detalle el problema de planificación, y con ello poder encontrar una solución adecuada.
- *Conocer el funcionamiento del algoritmo Backfill*: Es necesario conocer cómo se comporta en profundidad el algoritmo Backfill, y cuáles con sus principales virtudes y defectos, para poder realizar una correcta implementación.

- *Análisis de IRMaSim*: Hay que ser consciente de la forma en la que está implementada la herramienta para entender cómo funciona, y de esta forma ser capaz de conocer dónde se deben aplicar los cambios para mejorar su funcionamiento.
- *Diseño e implementación de objetivos*: Se realizan sucesivas versiones del algoritmo hasta alcanzar la implementación correcta, puliendo los diferentes problemas que iban surgiendo durante el desarrollo.
- *Comprobación de las soluciones*: Una vez se tiene la implementación completamente funcional, se realizan una serie de experimentos en la herramienta para validar la solución y verificar que concuerda con el funcionamiento real.
- *Depuración y optimización*: Con la implementación correcta y los experimentos validados, se trata de realizar las posibles modificaciones en aras de optimizar su rendimiento, precisión, etc.

1.4. Estructura del Documento

El documento presentado consta de cinco capítulos incluyendo el presente *Capítulo 1* de *Introducción*.

- *Capítulo 2*: Se presentan los conceptos básicos de la planificación de datracenters y las características fundamentales de IRMaSim.
- *Capítulo 3*: Se describen los cambios que se han realizado en el simulador, incluyendo el diseño e implementación del algoritmo Backfill para la mejora de la planificación de tareas en IRMaSim.
- *Capítulo 4*: En este capítulo se indican y detallan los experimentos realizados para comprobar que la implementación del algoritmo es correcta. Se comienza con una prueba sencilla para, a continuación aumentar la exigencia de los ensayos. Estos experimentos son detallados de manera exacta, para que cualquier persona pueda realizarlos y obtenga los resultados descritos.
- *Capítulo 5*: Para finalizar el proyecto se presentan las conclusiones finales, así como los posibles vías de mejora del simulador IRMaSim.

Capítulo 2

Background

En este capítulo se presentan una serie de conceptos que son importantes para la correcta comprensión del documento. En primer lugar se describen algunas palabras importantes relacionadas con el proyecto. A continuación se explica brevemente en qué consiste la planificación Backfill, en la que se basa el presente proyecto. Después se hace un repaso acerca de la herramienta empleada para desarrollar el proyecto (IRMaSim). Y por último, se muestran una serie de trabajos de otros autores, que tienen una estrecha relación con el desarrollo de este.

2.1. Glosario

En esta sección se van a desgranar algunos de los términos clave del desarrollo del trabajo, lo que facilitará su interpretación y comprensión a lo largo del documento. Son conceptos relacionados con la planificación y ejecución de tareas en Datacenters.

- *Job*: Cada uno de los trabajos que llegan para ser ejecutados por el datacenter.
- *Id*: Elemento identificativo de cada job.
- *Subtime*: Tiempo de llegada del job a la cola de planificación.
- *Res*: Recursos requeridos por el job como por ejemplo, número de procesadores o ancho de banda de memoria.
- *Estimated Execution Time (a partir de ahora EET)*: Tiempo máximo de ejecución de la tarea, introducido por el usuario como techo.
- *Req_ops*: Operaciones necesarias a ejecutar por el core para completar el job.
- *IPC*: Instrucciones ejecutadas por ciclo de CPU para ese job específico.

- *Mem*: Cantidad de memoria que necesita un job para poder ejecutarse. (MB)
- *Mem_vol*: Ancho de banda de memoria. (GB/s)
- *Time-Step*: Cada uno de los pasos por los que pasa el simulador durante la ejecución y ocurre algún evento.

2.2. Algoritmo de Planificación Backfill

Hoy en día, el procesamiento de datos y las tareas a ejecutar por parte de los datacenters está en auge. Debido al gran incremento tanto en el número como en la variedad de las aplicaciones que hacen uso de las infraestructuras de los datacenters, su uso se ha incrementado notablemente, y esto continuará al alza en los próximos años. El impacto que puede tener sobre el rendimiento y la eficiencia, la forma en la que se ejecutan los diferentes trabajos, hace que este campo sea tenido muy en cuenta en lo que se refiere a la investigación y el desarrollo.

El correcto funcionamiento de un Datacenter depende de un gran número de factores (Hardware disponible, consumo energético, sistemas de ventilación, etc), pero entre ellos destaca la planificación de las tareas a ejecutar. Este es un tema clave ya que puede optimizar el uso de los componentes que hay disponibles. En función de la política que se escoja, los trabajos entrarán al sistema de forma diferente, lo que hace que puedan adecuarse en mayor o menor medida a los recursos del sistema. Una mala elección podría hacer que se sobredimensionasen los medios para un determinado job, lo que implicaría un uso ineficiente del sistema.

A priori, el algoritmo de planificación más sencillo que se nos puede ocurrir es *FIFO*, en el que se van ejecutando los jobs en el mismo orden en el que van llegando. Su gran baza es la sencillez, ya que no requiere de mucha inteligencia por detrás para su implementación, pero precisamente por esto, resulta extremadamente ineficiente en términos de uso en un datacenter real cuya carga es considerable. Backfilling [12] surge como una alternativa para tratar de maximizar la eficiencia con la que utilizan los recursos del sistema.

Tradicionalmente, con *FIFO*, cuando el job que está en la cabeza intenta ejecutar y no tiene recursos disponibles, toda la cola se ve bloqueada, por lo que la finalización de las tareas se ve demorada.

Backfill consiste en tratar de aprovechar las ventanas temporales en las que un job está bloqueado a la espera de tener recursos disponibles. En el momento en que el job es bloqueado, los recursos que están libres son reservados, por lo que solo pueden hacer uso de ellos los trabajos que entren a ejecutar debido al backfill. Si esto no ocurriera, podría entrar a ejecutar un job que necesite menos recursos y cuyo *EET* sea muy elevado, causando un problema de inanición en el job bloqueado. Además, cuando el job se bloquea, se calcula cuándo va a tener disponible el/los recursos necesarios para que pueda planificarse. Con ese tiempo,

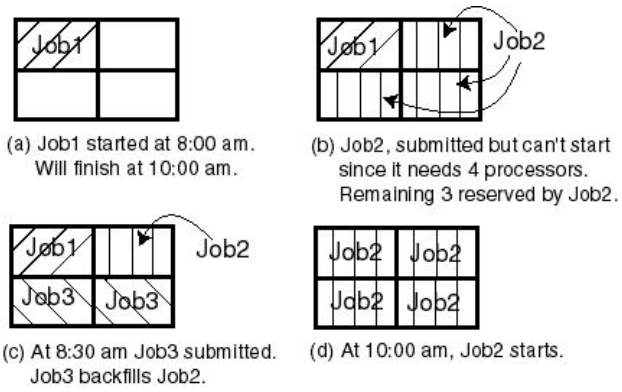

Figura 2.1: Algoritmo Backfilling[1]

se realiza una comprobación de las tareas que están pendientes de entrar a ejecutar en el sistema y, se produce backfill, adelantando la ejecución, cuando:

- El job necesita los mismos o menos recursos que el job bloqueado
- Su *EET* es inferior al tiempo calculado en que el job bloqueado tendrá sus recursos disponibles.

En la figura 2.1 podemos ver un pequeño ejemplo de como se comporta este algoritmo con tres jobs. Tenemos el *Job1* ejecutando en el sistema, haciendo uso de 1 core y cuyo *EET* son las 10:00 am. El *Job2* intenta ejecutar, pero no puede ya que necesita 4 cores y solo hay 3 disponibles en el sistema, por lo que es bloqueado. Momentos después el *Job3* entra al sistema y, como solo necesita 2 cores y su tiempo de finalización es anterior a las 10:00 am, se produce backfilling y ejecuta. Para las 10:00 am, el *Job2* tiene todos sus recursos disponibles así que ejecuta con normalidad.

En este tipo de planificador, el factor humano juega un papel crucial ya que en función de los parámetros que decida incluir el usuario en cada uno de los jobs, el rendimiento se verá incrementado o disminuido. El usuario debe indicar cuál es el *EET* y, según como de buena sea la aproximación de ese tiempo al comportamiento real del modelo, las ganancias de rendimiento serán mayores. En el caso de hacer unas malas aproximaciones, lo que ocurre es que no se producirá nunca el adelantamiento de los jobs y por tanto como cota inferior, el algoritmo se comportará como *FIFO*.

Una vez establecido el *EET*, pueden ocurrir tres cosas en comparación con la ejecución real. Si el job finaliza de acuerdo al *EET*, todo funciona según lo previsto, por lo que se continúa con la planificación normal. El job puede finalizar antes que el tiempo estimado, por lo que la ventana temporal se reduce, y el job bloqueado dispone de los recursos necesarios antes. En este caso, el job activo puede finalizar, pero el job bloqueado no dispone necesariamente de todos los recursos por lo que también podría considerarse una nueva planificación en *Backfill*, adelantando el siguiente trabajo de la cola. La última posibilidad es que el job tarde más en

ejecutar de lo que había previsto el usuario. En este caso, el job es expulsado para dar paso al job que estaba bloqueado.

Por esta razón es muy importante el tiempo estimado de finalización, ya que debe tomarse siempre como una cota superior de ejecución, o de lo contrario, no se puede tener certeza de que el conjunto de jobs vaya a ser ejecutado de forma correcta [12].

2.3. IRMaSim

El lugar idóneo de desarrollo de este tipo de proyectos es un datacenter real. De esa forma los trabajos se ejecutan en un entorno efectivo y, los resultados y métricas obtenidos se ajustan más a la realidad. El problema viene cuando la carga de trabajo es tan grande que las pruebas a realizar requieren varios días o incluso semanas de ejecución para finalizar, con el consiguiente gasto de energía, que puede llegar a ser considerablemente alto.

Debido a esto nace IRMaSim como una herramienta que permite realizar simulaciones con una tasa de errores muy baja, y que consigue speedups altos en comparación con la ejecución real [11].

IRMaSim surge en el *Grupo de Arquitectura y Tecnología de Computadores de la Universidad de Cantabria* y se caracteriza por representar de una forma bastante fiel el comportamiento de un datacenter real. Organiza los recursos computacionales de forma jerárquica para así poder representar ordenadores multiprocesador y multicore, pudiendo de esta forma asignar parámetros individuales a cada elemento (frecuencia de reloj, memoria, consumo energético, etc).

El objetivo es disponer de un simulador que consiga un buen equilibrio entre la velocidad y la precisión. Esto resulta vital para conseguir buenos resultados, lanzando diferentes experimentos de forma ágil, aproximándose en gran medida a los logros obtenidos en un cluster real gestionado por ejemplo con *SLURM*.

El simulador toma en consideración diferentes parámetros que van más allá de le ejecución de las tareas, como pueden ser el rendimiento, el consumo energético o la contención que pueda ocurrir en el acceso a memoria.

En cuanto a la planificación, la herramienta incluye una serie de políticas que es posible emplear, siendo la más interesante la basada en *Deep Reinforcement Learning* que mediante el entrenamiento de una red neuronal es capaz de asignar los jobs, en base a su naturaleza, a los recursos que más le convienen.

La realidad es que los grandes centros de procesamiento de datos no son homogéneos, es decir, cuando se crean sí que pueden tener todos los componentes las mismas características, pero según van pasando los años y nuevas tecnologías van surgiendo, no se cambian todas

las piezas de golpe si no que se tiende a integrar los nuevos componentes en la estructura ya operativa.

Esto hace que se tengan recursos computacionales de distinta índole, cuyas capacidades en cuanto a poder de cómputo, memoria, o eficiencia energética difieren. Asignar los diferentes jobs a los diversos recursos es una tarea muy importante por lo que el papel que juega el planificador en un datacenter resulta clave para alcanzar una buena usabilidad y rendimiento del sistema [13]. Como podemos observar en la Figura 2.2, un cluster real está formado por distintos nodos, y no todos ellos tienen por qué ser iguales, si no que conviven elementos de diferentes capacidades y arquitecturas.

De hecho hoy en día esta es una tendencia habitual a seguir por la mayoría de fabricantes. Se combinan diferentes tipos de procesadores y cores, mezclando unos de mayor potencia y consumo con otros no tan potentes y con un consumo menor, en aras de mejorar la eficiencia energética sin perjudicar el rendimiento. Este punto de vista tiene mucho sentido ya que no es necesario emplear los elementos más potentes del sistema para ejecutar tareas en segundo plano o con una menor carga [14].

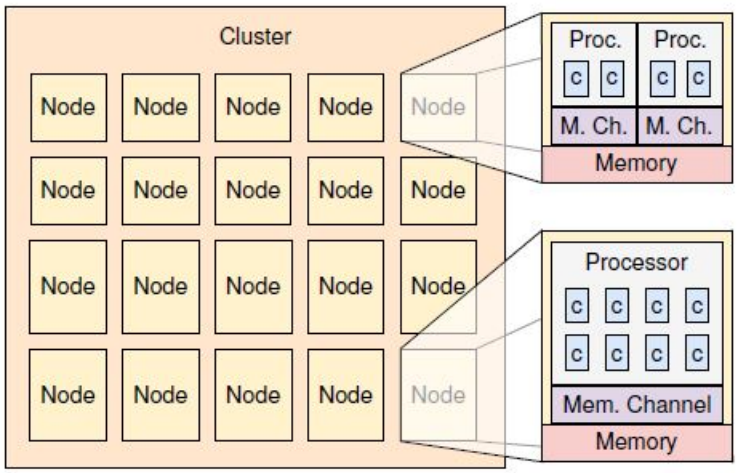

Figura 2.2: Cluster Heterogéneo[2]

La herramienta tiene esto en cuenta, y a la hora de seleccionar la plataforma se especifican cuáles son sus características exactas. Toda la configuración general, así como la descripción del hardware o la naturaleza de los jobs se especifica mediante los siguientes ficheros JSON, que configuran la entrada del simulador:

- *options.json*: En este fichero se indican aspectos generales que son empleados por la simulación, como pueden ser, el nombre de la plataforma, el entorno y el agente para predicciones basadas en inteligencia artificial, o el nombre de los ficheros de descripción del hardware y de los jobs.
- *platform.json*: Este fichero representa la plataforma sobre la que se van a lanzar los trabajos a ejecutar. El esquema es bastante fiel a la realidad ya que podemos indicar

atributos como por ejemplo los clusters, los nodos, los procesasdores, las conexiones locales, etc.

- *jobs.json*: Este fichero se emplea como una descripción de los jobs a ejecutar por la plataforma. Cada job viene dado por un identificador, su tiempo de llegada, el número de recursos que necesita, un perfil(con requerimientos de operaciones y memoria) y el *EET*

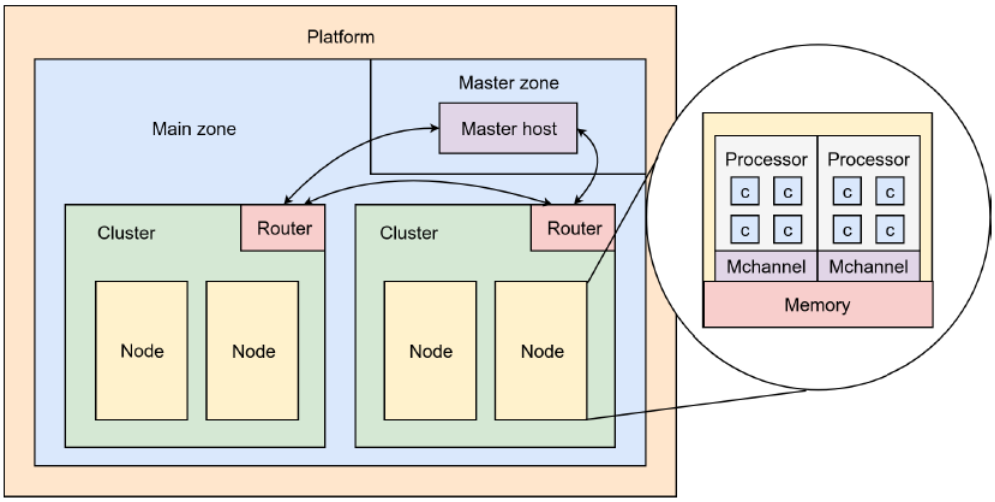

Figura 2.3: Representación del datacenter en IRMaSim [1]

En cuanto a la estructura del datacenter para IRMaSim podemos distinguir los siguientes elementos:

- *Platform*: Componente principal del sistema.
- *Main zone*: Zona en la que se encuentran los recursos del datacenter.
- *Master zone*: Zona en la que se encuentra el master host.
- *Master Host*: Elemento encargado de realizar la planificación de tareas.
- *Cluster*: Cada una de las piezas que conforman el datacenter. Formado por un conjunto de nodos y un router que se encarga de las comunicaciones.
- *Nodo*: Partes de cómputo del cluster formadas por un conjunto de procesadores.
- *Procesador*: Pieza de cómputo del nodo. Formado por un conjunto de cores.
- *Core*: Unidad de cómputo del sistema. Encargados de realizar la ejecución efectiva de las tareas.
- *Memoria*: Cada nodo tiene su propia memoria, y esta es compartida por cada uno de sus procesadores y cores.

Como podemos observar en la Figura 2.3 la memoria de cada nodo es compartida por todos los procesadores, y a su vez por los cores de cada uno de estos. Esto tiene una alta importancia a la hora de ejecutar las tareas ya que en función del volumen de memoria empleado por cada job, si se planifican varios jobs en el mismo nodo de cómputo, y el volumen agregado supera el del procesador, la ejecución se verá ralentizada.

2.4. Trabajos Relacionados

A la hora de planificar tareas en datacenters no existe un algoritmo que sea el idóneo para todas las ocasiones. Esto se debe a que la planificación en sí es un problema NP-completo, y por lo tanto es necesario el uso de algoritmos heurísticos para resolverla, lo que conlleva que ninguno de los empleados sea óptimo. En función del objetivo para el que esté diseñado se puede emplear una política u otra. Todos los algoritmos tienen algún déficit que es corregido por otro algoritmo, pero este a su vez presenta otro tipo de deficiencia, lo que hace que la elección a seguir sea de vital importancia [7].

Partiendo desde los algoritmos más tradicionales, nos encontramos algunos como el ya mencionado *FIFO*, en el que los jobs se van ejecutando en orden de llegada, o el *SJF* en el que el job seleccionado para ejecutar es el que tiene un menor tiempo de ejecución [15].

Los algoritmos de planificación, además, pueden ser expulsores como en el caso de *Round Robin*, que va rotando los jobs en ejecución para tratar de maximizar el *fairness* y conseguir así que todos estén activos una parte proporcional del tiempo [15]. Esto evita que un único proceso monopolice el consumo de los recursos.

Estos algoritmos son funcionales pero no logran exprimir al máximo el sistema. Todo parece apuntar al uso de la inteligencia artificial para lograr una planificación más efectiva. Una propuesta de esto es el trabajo *Gestión de recursos en Datacenters basado en Deep Reinforcement Learning* [16] en el que se emplea un agente que se va retroalimentando en cada paso de la simulación para aprender y mejorar, y mediante una red neuronal decide que tarea es la óptima a ejecutar en cada caso.

A raíz del uso de inteligencia artificial para planificación de datacenters, el artículo *Deepjs: Job scheduling based on deep reinforcement learning in cloud data center*[17] propone un contrapunto a los algoritmos heurísticos tradicionales. Mediante técnicas de *Deep Reinforcement Learning* se hace uso de una función *fitness* que trata de minimizar el tiempo de finalización de los jobs, maximizando el throughput.

Otro proyecto muy interesante resulta *Preemptive Data Center Scheduling Without Runtime Estimates*[18] en el que se propone realizar una planificación multinivel, debido a la topología de los dataenters, y en la cual un elemento central distribuye los jobs sobre los distintos nodos para, después en cada nodo realizar la planificación sobre los cores teniendo en cuenta el histórico de ejecución de los jobs.

Este tipo de mejora también se propone para el algoritmo backfill en *Backfilling Using System-Generated Predictions Rather than User Runtime Estimates* [12]. Se plantea hacer uso del histórico de los jobs que ya han ejecutado para mediante técnicas de aprendizaje reforzado, poder predecir cuál va a ser el *EET* y de esta forma realizar una planificación más ajustada a la realidad. Con este método el usuario deja de ser el que indica cuál es ese valor, si no que es el propio sistema el que lo proporciona.

Un punto de vista a tener en cuenta es el que se presenta en *Green IT scheduling for data center powered with renewable energy*[19], en el que optan por focalizar más en la eficiencia energética que en el rendimiento bruto. Para ello proponen una planificación cuyo heurístico viene alimentado por un sistema de gestión de energías renovables. Los jobs son ejecutados en bloques y se establece una fecha límite que es la del fin de existencias de energía verde. De esta forma se maximiza el uso de energía limpia, dejando la utilización de energía marrón en segundo plano.

Por último, en *E-OSched: a load balancing scheduler for heterogeneous multicores* [20], se realiza un estudio de como se distribuye la utilización del sistema en plataformas heterogéneas compuestas por CPUs y GPUs. Se observa que la tendencia es, mediante OpenCL como lenguaje de programación, hacer un uso excesivo de las capacidades de las GPUs, mientras que las CPUs quedan prácticamente inutilizadas. Por ello se desarrolla E-OSched, cuyo propósito es realizar un balanceo de carga entre ambos elementos y así lograr un menor tiempo de ejecución y un mayor throughput.

Capítulo 3

Diseño e Implementación del Algoritmo Backfill en IRMASim

En este capítulo se describe la contribución de este Trabajo Fin de Grado sobre IRMaSim. Para ello se detallan los procesos de diseño e implementación del algoritmo necesarios para incluirlo en la herramienta y ser capaz de planificar tareas con independencia del tamaño y tipo de workload o plataforma.

3.1. Análisis de requisitos

Para comenzar este capítulo se va a realizar un pequeño análisis de los requisitos a valorar para el desarrollo de este proyecto. No solo se debe tener en cuanta que su funcionamiento sea adecuado, si no que también se deben tener en cuenta métricas como el rendimiento o la eficiencia.

3.1.1. Requisitos funcionales

Los requisitos funcionales, por su definición, se trata de aquellos que describen cualquier actividad que el sistema deba realizar, o dicho en otras palabras, la función que debe realizar el sistema cuando se cumplen una serie de condiciones [21].

En este caso, el requisito funcional prioritario y más claro es que la herramienta debe ser capaz de realizar la planificación de tareas empleando el algoritmo *Backfill*. La forma en la que se seleccionan los trabajos, y se van moviendo entre las diferentes colas del sistema debe ser la adecuada. Cuando finalice la simulación, el orden en el que se ejecutaron los distintos trabajos debe poder ser comprobado a través de los ficheros de log.

Además, a medida que va avanzando la simulación, se podrá ver por terminal cuando un job es bloqueado y por tanto entra en juego el beneficio de *Backfill*, calculando la ventana de la que disponen los demás trabajos para poder adelantar su ejecución y mostrándola por pantalla.

3.1.2. Requisitos no funcionales

En cuanto a los requisitos no funcionales, estos se definen como atributos que pueden utilizarse para juzgar el comportamiento del sistema en lugar de su operación específica [22].

Teniendo esta definición en mente, los principales requisitos no funcionales son los siguientes:

- *Rendimiento*: La herramienta debe ser capaz de realizar la planificación en un tiempo asumible. Su propósito principal es sustituir la ejecución real por la simulada, obteniendo buenos resultados pero en un tiempo sensiblemente menor, ya que si no, no tendría sentido ejecutar IRMaSim. Como se verá en el Capítulo 4, se realizan simulaciones de minutos en apenas segundos.
- *Precisión*: Los resultados obtenidos deben ser similares, en gran medida, a los que obtendríamos en la ejecución real. IRMaSim asume un porcentaje de error de entorno a un 5 % como máximo, lo cual para trazas grandes resulta prácticamente insignificante.
- *Robustez*: El sistema debe ser operativo independientemente de cuáles sean los parámetros de entrada recibidos. En un sistema real, se puede dar cualquier combinación de plataforma y conjunto de tareas por lo que la herramienta debe estar preparada para tratar con todas ellas.
- *Escalabilidad*: Todos los cambios que se realicen en la herramienta deben ser compatibles con las características con las que ya cuenta el sistema. El hecho de incorporar una política de planificación adicional no debe hacer fallar las planificaciones que ya existen. Igualmente, los cambios deben diseñarse con vistas a que la herramienta va a seguir evolucionando y por tanto también debe ser compatible con versiones futuras.

Si se cumplen todos los puntos descritos en esta sección, nos aseguraremos un correcto desarrollo de software que funcione de forma eficaz y sea capaz de dar una respuesta satisfactoria al problema planteado inicialmente.

3.2. Decisiones de diseño

Internamente, IRMaSim está formado por un conjunto de ficheros Python. Se estructura de tal forma que tenemos un elemento principal que es el Simulador, el cual a su vez contiene los tres componentes core de la herramienta.

Esta estructura se puede apreciar en la figura 3.1. En ella podemos observar las diferentes partes de las que está compuesta la herramienta, así como los elementos de entrada. En la imagen, los elementos en color verde son los que se han desarrollado en este TFG, los azules los que ya estaban, y los que tienen un degradado azul y verde son elementos que ya existían pero han sido modificados en este trabajo, bien para optimizarlos o bien por necesidad de la nueva implementación.

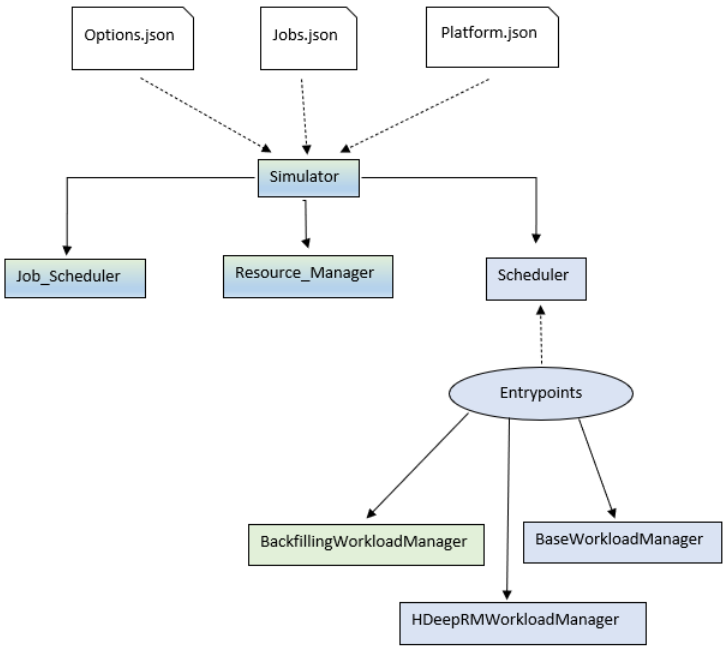

Figura 3.1: Estructura del simulador IRMaSim

Respecto a los componentes principales, podemos observar en la figura 3.1 que se diferencian tres partes:

- *Job_Scheduler*: Componente que se encarga de tomar en consideración todos los jobs que son cargados en el sistema. Está formado por un conjunto de colas por las que van pasando los trabajos en las diferentes etapas de su ciclo de vida. Lleva la cuenta de qué jobs están corriendo, cuáles pendientes o si existe alguno bloqueado.
- *Resource_Manager*: Componente que se encarga de gestionar todos los recursos computacionales del sistema. Dispone de una visión completa de la plataforma y, asigna y libera los recursos necesarios de cada job.
- *Scheduler*: Componente sobre el que recae toda la inteligencia de la planificación de tareas. Decide cómo deben ejecutar los jobs en el sistema.

También se observan los puntos de entrada, de los que se habló en la descripción de la herramienta IRMaSim en la Sección 2.3. Esta herramienta recibe el fichero options.json, con

características generales, el fichero jobs.json, con la descripción del conjunto de jobs, y el fichero platform.json, con la descripción de los elementos que componen la plataforma.

En cuanto al scheduler, las diferentes implementaciones disponibles se encuentran en el directorio *Entrypoints*. En él, encontramos los planificadores que vienen por defecto en la herramienta, que son el Base y el HDeepRM. El primero de estos emplea un algoritmo FIFO sencillo y el segundo utiliza *Deep Reinforcement Learning* a través de un agente y un entorno para decidir cuál es el job a ejecutar.

Este proyecto se basa principalmente en la inclusión de un nuevo tipo de planificador en ese directorio, el *BackfillingWorkloadManager*.

3.3. Implementación del algoritmo Backfill

En esta sección se describe con detalle la implementación del algoritmo *Backfill* en el simulador IRMaSim (ver Sección 2.3). El algoritmo se describe de forma teórica en la Sección 2.2, pero como veremos la implementación tiene algunas peculiaridades.

Los pasos fundamentales a seguir, a modo de pseudocódigo, son los siguientes:

- Se selecciona el primer job de la lista de trabajos pendientes.
- Se comprueba si dispone de los recursos necesarios para ejecutar.
- En caso de tener los recursos necesarios, ese job ejecuta y se trata de planificar el siguiente en la lista.
- En caso de no disponer de los recursos, se calcula el *Backfilling Gap*, es decir, el intervalo del tiempo en el que con total seguridad el job será capaz de ejecutar.
- Se itera en la lista de trabajos pendientes para ver si existe algún job que tenga los recursos necesarios, y su *EET* sea inferior al *Backfilling Gap* calculado.
- Si existe, adelanta su ejecución

Estas son, de forma somera, las operaciones que se deben realizar en cada paso de la simulación. Esta serie de instrucciones son repetidas hasta que ya no quedan más trabajos que planificar, por lo que el *Backfilling Gap* se va recalculando constantemente para dar la posibilidad a nuevos jobs entrantes de ejecutar si cumplen los requisitos.

El algoritmo en sí no define formalmente el *Estimated Execution Time, EET* en la literatura, por lo que se puede interpretar de diferentes formas. Las dos lecturas posibles que tiene son:

- El EET es el tiempo absoluto para el que ese job ya debe haber terminado. Por ejemplo, las 13:00.
- El EET es la duración estimada de ejecución del job. Por ejemplo, 2 horas.

En ambos casos, el dato aportado siempre se toma como techo, por lo que, para el momento indicado, la tarea debe haber finalizado bajo cualquier circunstancia, o de lo contrario el job es expulsado del sistema para dar paso al siguiente trabajo.

En este trabajo se ha optado por la segunda acepción del *EET*. En la clase que modela el Job, se añaden dos nuevos parámetros a los ya existentes. Se trata del propio *EET*, y el tiempo estimado de finalización, que en el momento de creación del job es desconocido, y es calculado en el momento que el trabajo entra a ejecutar en el sistema. Se calcula haciendo la suma del *EET* y el momento en el que se encuentra la simulación cuando el job es elegido para ejecutar, proveniente de la cola de trabajos pendientes.

Atributos	**Tipo**
Id	int
Name	string
Type	string
Profile	dict
Subtime	float
Resources	int
Req_ops	float (Profile['req_ops'] / Profile['ipc'])
IPC	int (Profile['ipc'])
Mem	int (Profile['mem'])
Mem_vol	int (Profile['mem_vol'])
Allocation	array
Core_finish	array
Estimated_execution_time	float
Estimated_finish_time	float

Cuadro 3.1: Atributos de los jobs para IRMaSim

En el Cuadro 3.1 podemos observar los diferentes atributos de los que consta un Job en la herramienta, así como sus tipos de datos. Entre los más destacados podemos encontrar el identificador, el subtime, los recursos requeridos, las necesidades de memoria, o los nuevos elementos necesarios para la planificación *Backfilling*. Los elementos que más afectan para la implementación del algoritmo vienen detallados en la Sección 2.1.

En la implementación en Python, la clase está compuesta de un conjunto de funciones para modelar el comportamiento descrito anteriormente.

En primer lugar tenemos el constructor de la clase, al cual se le pasan como parámetro un diccionario de opciones y un objeto simulador de la clase Simulador. El diccionario se emplea para diferentes parámetros de configuración como pueden ser el fichero de jobs, la plataforma o el fichero de escritura de los resultados, y el simulador se emplea para interactuar tanto con el *Resource_Manager* como con el *Job_Scheduler*.

Adicionalmente disponemos de cuatro funciones empleadas a modo de señales. Estas son utilizadas para que cuando ocurra algo en el sistema que afecte de algún modo al planificador, este sea alertado y consciente de lo que está ocurriendo. Las cuatro señales son:

- *onJobSubmission*: Señal que se activa cuando se incorpora un nuevo job a la plataforma, más específicamente a la cola de jobs en espera de ejecutar (su submit time coincide con el tiempo del simulador). En este caso, simplemente registra la llegada del job en un fichero de log. Recibe como parámetro el job en cuestión.

- *onJobCompletion*: Señal que se activa cuando un job finaliza su ejecución. Realiza la liberación de los recursos empleados, y ajusta la cuenta de trabajos activos y completados. Recibe como parámetro el job finalizado.

- *onNoMoreEvents*: Esta señal se activa cuando no existen más eventos para un determinado *Time-Step*. Si ya se han cargado todos los jobs nuevos en el sistema, y los que han finalizado han sido liberados, se usa esta señal para indicar al scheduler que realice una nueva planificación con los jobs disponibles.

- *onSimulationEnds*: Señal activada cuando finaliza la simulación. Se encarga de escribir los resultados en el fichero indicado en el diccionario de opciones.

Por último, se desarrolla la función *schedule_jobs*, que es la más larga y compleja de todas, por lo que la mayor parte del esfuerzo reside aquí para que funcione correctamente. Se trata de la implementación del algoritmo en lenguaje Python. Esta función es llamada en cada *Time-Step* de la simulación y se encarga de decidir qué jobs ejecutan, con qué recursos y en qué orden.

La implementación detallada del algoritmo se describe a continuación.

En primer lugar, se define una variable booleana llamada "Serviceable"que nos indica si podemos seguir planificando más jobs en ese *Time-Step*, o por el contrario, con los recursos disponibles ningún job más puede entrar a ejecutar en el sistema.

A continuación nos encontramos un bucle "while.[en] el que se itera mientras queden trabajos pendientes y no se hayan agotado los recursos del sistema. En cada iteración se selecciona el primer job de la cola de trabajos pendientes y se consulta al *Resource_Manager* si se le pueden asignar recursos. Si el job tiene los recursos necesarios, estos son asignados y se añade la tarea a una lista de jobs planificados, eliminándolo de la cola de pendientes.

En caso de no disponer de los recursos suficientes, se comprueba que existen jobs corriendo en el sistema para así poder calcular el *Backfilling Gap*. Si no existen, se asigna la variable "Serviceable.ª false y el step de planificación finaliza. Si existen, se recorre de nuevo la cola de jobs pendientes en busca de uno que satisfaga las necesidades de *Backfilling*.

Se comprueban todos los trabajos y, si existe alguno cuyo tiempo de finalización estimado es menor que el *Backfilling Gap*, y además dispone de los recursos necesarios se añade a la lista de jobs planificados, eliminándolo también de la cola de pendientes. Si ninguno de los jobs restantes son aptos para Backfill, el valor de la variable "Serviceable"se asigna a false igualmente.

Por último, se añade a la cuenta de trabajos activos y corriendo, el número de jobs planificados en este *Time-Step*, y se indica al *Job_Scheduler* que esos jobs pasan de estar pendientes a estar en ejecución.

3.4. Solución de errores

Como es normal en cualquier proyecto, no funciona todo a la primera como nosotros deseamos, y a medida que se iba desarrollando el código, iban surgiendo errores inesperados que no estaban contemplados al comienzo del proyecto.

En esta sección se va a hacer un repaso acerca de los errores más destacados al realizar avances en el trabajo, y se dará una detallada explicación de la solución propuesta para resolver cada uno de ellos.

En primer lugar, se observó un error en el comportamiento del algoritmo. En las primeras versiones del programa, se almacenaba como atributo de la clase *BackfillingWorkloadSimulator* tanto el backfilling gap como el id del job bloqueado. Estos valores eran actualizados dentro de la función *schedule_jobs* cuando un job no disponía de los recursos necesarios para ejecutar. Se calculaba el backfilling gap y se asignaba al atributo de la clase, además de guardar el identificador del job que era bloqueado. Estos valores eran empleados para realizar las comparaciones con los demás jobs, y únicamente volvían a cambiar su valor una vez que el job que estaba bloqueado pasaba a ser planificado, independientemente de los ciclos de simulación que hubieran transcurrido.

Tras varias ejecuciones del programa nos dimos cuenta de que este comportamiento no era correcto. En ocasiones ocurría que, cuando el job bloqueado dependía de varios jobs, cuando uno finalizaba la ventana de Backfill podría reducirse, pero como el nuevo *Backfilling Gap* no se recalculaba con ese evento, el planificador no se daba cuenta de ello. En este caso, la ventana únicamente puede reducirse, haciendo que el trabajo bloqueado disponga de sus recursos antes de lo estimado, y no aumentarse, ya que el job que causaría ese aumento, no debería haber podido entrar a ejecutar en el sistema.

La solución a este problema es no almacenar ningún atributo referido al Backfill en la clase, y hacer que el *Backfilling Gap* sea recalculado en cada paso de la simulación. Esto hace que cada evento que ocurra en el sistema sea tenido en cuenta, y por lo tanto es susceptible de modificar el comportamiento de la planificación.

Esto se debe a que el tiempo estimado de ejecución, aportado por el usuario no es completamente preciso y, por lo general las tareas tienden a finalizar antes que el *EET* dado.

Este comportamiento se puede apreciar con un simple ejemplo de cuatro jobs en una plataforma de dos cores, detallado en la Sección 4.2.1.

Otro error que modificaba el comportamiento del algoritmo venía dado por la estructura de datos empleada para el manejo de las listas de jobs.

Para la implementación del planificador utilizando inteligencia artificial(HDeepRM) se emplea la estructura de datos heap para almacenar los jobs leídos del fichero *jobs.json*. Durante la simulación, la inserción y extracción de las diferentes colas se hace mediante los métodos push y pop, propios de la estructura de datos de tipo cola. Push añade un elemento al final de la cola, y pop extrae la cabeza de la cola.

En el caso del planificador HDeepRM, el funcionamiento es correcto independientemente de la estructura de datos empleada, con tal que de que la cabeza de la cola sea la correcta, por lo que se emplea el heap ya que tiene propiedades logarítmicas en cuanto a complejidad computacional. Heap garantiza que la cabeza siempre es el elemento con el mínimo valor(en este caso subtime), pero no proporciona ninguna seguridad en cuanto al orden de los demás elementos ya que los va recolocando en cada inserción o extracción

```python
import heapq

H = [21,1,45,78,3,5]
# Create the heap

heapq.heapify(H)
print(H)

# Remove element from the heap
heapq.heappop(H)

print(H)
```

Figura 3.2: Ejemplo de uso de la estructura de datos Heap

En la figura 3.2 se muestra un pequeño ejemplo del funcionamiento de heap. Comenzamos creando un array con los elementos que queremos y usamos el método heapify para convertirlo en un heap. En el primer print, el array se ordena de la siguiente forma: [1, 3, 5, 78, 21, 45]. Tras emplear el método pop, el elemento 1 es eliminado del array, y el segundo print queda de la siguiente manera: [3, 21, 5, 78, 45].

Para el planificador *Backfill*, esto no es deseable que ocurra ya que cuando tenemos jobs con el mismo subtime estos son desordenados, pero nosotros queremos utilizar el mismo orden exactamente que en el que son leídos del fichero de jobs.

Para solucionarlo, he decidido emplear directamente el array sin transformar y asegurarme que las inserciones y extracciones de la estructura de datos siempre se hacen en el orden correcto.

Por último, cabe destacar un error que afecta a la primera iteración de la planificación para un grupo de tareas dado. En el caso de disponer de una plataforma con múltiples cores, independientemente de cuál sea su tamaño, y un conjunto de tareas cuya suma de recursos exceda la capacidad de la plataforma, si múltiples trabajos coinciden con el mismo submit time, no se podrá realizar *Backfill* en ese time step.

Esto se debe a que para buscar el backfill gap y determinar cuando se van a liberar los recursos necesarios, se requiere disponer de trabajos en ejecución en el sistema, puesto que la comprobación se realiza con la cola de trabajos activos.

En la primera iteración se planifican jobs hasta que ya no quedan recursos para el último, y es bloqueado. En ese momento finaliza el time step y los trabajos planificados son lanzados a ejecutar, y es en los siguientes cuando ya si que se puede realizar *Backfill* al disponer la plataforma de trabajos corriendo.

3.5. Otras mejoras propuestas en la herramienta

Adicionalmente a la propia implementación del algoritmo, se han diseñado algunos otros métodos que son útiles para el uso de la herramienta, o modificado alguno ya existente para perfeccionar su funcionamiento.

Es el caso de la Función 1, que pertenece a la clase *Job_Scheduler* y la cual trata de encontrar el tiempo mínimo en el que el job bloqueado dispondrá de los recursos necesarios. Internamente, se realiza una copia de los jobs que están actualmente corriendo y se reordenan de acuerdo a su tiempo estimado de finalización. Una vez ordenados, se itera por ellos hasta obtener que el agregado de los recursos de los jobs es igual o superior a los necesitados (introducidos como parámetro). El valor retornado corresponde al tiempo de finalización estimado del último job que satisface la necesidad.

```python
def backfilling_best_option(self, needed_resources: int) -> int:

    # Recorre los jobs en ejecución buscando el mejor tiempo para Backfilling
    print("Necesito:", needed_resources)
    options = self.jobs_running.copy()
    print("Jobs corriendo:", self.nb_running_jobs)
    options.sort(key=lambda x: x.estimated_finish_time, reverse=False)
    res = 0
    i = 0
    while res < needed_resources:
        print("Job:", options[i].id, "tiene", options[i].resources)
        res += options[i].resources
        i += 1
    return options[i-1].estimated_finish_time
```

Función 1: Busca el mejor tiempo posible para Backfill

En la Función 2 se puede ver una función auxiliar creada para contabilizar los recursos libres de los que dispone, en un momento dado, el sistema. En ella simplemente se itera por la colección de cores de la plataforma y se tienen en cuenta aquellos cuyo estado no es sirviendo a un job.

```python
def available_resources(self) -> int:

    # Retorna los cores disponibles en el sistema
    return len([core for core in self.core_pool if not core.state['served_job']])
```

Función 2: Contabiliza los recursos libres del sistema

También se ha creado la función *find_init_time* en la clase *Simulator* que se encarga de realizar el cálculo entre los recursos necesarios, los recursos libres y los recursos empleados por otros jobs; o la función *get_N_pending_job* en la clase *Job_Scheduler* para obtener un trabajo en una determinada posición introducida como parámetro, lo cual antes no era posible.

En cuanto a rediseño de las funciones ya existentes, se ha corregido un error en la función *nb_running_jobs* que hacía que no se mostraran correctamente los jobs que estaban en ejecución en ese instante en el sistema. También la función *peek_job* ha sido modificada para ajustar su funcionamiento. A la hora de obtener un job de la cola de pendientes, la herramienta permite pasarle como opción una sorting_key que se encarga de elegir el orden en el que se almacenan los jobs. En la versión anterior su comportamiento no era del todo correcto, y ahora en caso de omitir la inclusión de la sorting_key, el job seleccionado es el primero de la cola.

Capítulo 4

Evaluación

En este capítulo se detallan los diferentes experimentos que se han realizado en la herramienta IRMaSim para probar el correcto funcionamiento de la planificación *Backill*, además de una comparativa con la planificación *FIFO*. Estos experimentos han sido realizados y diseñados para testear las aportaciones realizadas en el Capítulo 3. Los experimentos se explican de tal forma que cualquier persona es capaz de replicarlos y obtener los mismos resultados.

4.1. Metodología

En primer lugar, se debe definir correctamente la forma en la que se van a realizar los experimentos para corroborar que la implementación del algoritmo desarrollada en el Capítulo 3 es la adecuada. Esto resulta esencial, ya que de otra forma, realizando pruebas arbitrarias, los resultados que obtendríamos no serían fácilmente interpretables y por tanto no podríamos deducir unas conclusiones claras en cuanto a las virtudes del algoritmo planteado frente a los ya existentes.

Las distintas pruebas realizadas tienen como objetivo analizar el comportamiento de la herramienta, y la subyacente planificación en un número diferente de escenarios posibles, tratando de variar al máximo tanto la naturaleza y comportamiento de las tareas como la estructura de la plataforma hardware y así poder asemejarse más a un comportamiento real. Los experimentos se realizan con una complejidad creciente, partiendo de un ejemplo básico para comprender el funcionamiento de *Backfilling*, a un workload con una plataforma de tamaño reducido y un conjunto de jobs demandante.

Hay que tener en cuenta un factor fundamental. En el caso de IRMaSim, que es un simulador basado en trazas (estos son los ficheros de entrada con la definición de las tareas y los tiempos en los que se envían a la cola de ejecución), las simulaciones se lanzan ejecutando los jobs en lo que se conoce como *Procesamiento por Lotes*, ya que disponemos de un fichero con

todos los jobs al comienzo de la simulación, y estos, internamente van entrando al sistema de acuerdo al submit time. Esto se diferencia de un datacenter real, ya que lo habitual es que los trabajos vayan llegando, siendo ejecutados y expulsados de forma dinámica; sin conocer la dimensión de la traza desde el primer momento. Esto es algo a considerar ya que como veremos más adelante, la planificación de tipo *Backfill* será más apropiada para este último caso en el que hay un flujo constante de trabajos en el sistema.

Para que los experimentos sean adecuados se debe seguir un orden a la hora de su diseño, además de tener en cuenta distintos aspectos. Los pasos mostrados a continuación, describen el proceso de definición de los experimentos:

- *Definir el objetivo del experimento*: En este apartado se debe decidir cuáles van a ser las metas del experimento. Puede ser tanto que el orden en el que se ejecutan los jobs es el correcto de acuerdo al algoritmo, medidas de parámetros como el makespan (tiempo total de ejecución de la traza, desde que se lanza el primer trabajo hasta que finaliza el último), el consumo energético, el AWT (siglas de Average Waiting Time), etc. En función del objetivo que marquemos, se adaptarán los experimentos para conseguir profundizar en cada aspecto en una mayor o menor medida.

- *Diseñar la plataforma y el conjunto de jobs*: Una vez tenemos claro cuál es el objetivo de la prueba, se debe definir tanto la plataforma sobre la que se van a realizar los experimentos, como el conjunto de jobs que va a ser ejecutado.

  Respecto a los jobs se deben indicar las características recogidas en la tabla 3.1. Hay que analizar cuáles son los valores más apropiados para que el experimento se enfoque en la característica que deseamos evaluar. En cuanto a la plataforma, se deben definir varios aspectos que limitarán la capacidad del entorno. Entre ellos están el número de nodos de cada cluster, con su respectiva red de interconexión; los procesadores de cada nodo, su memoria y su ancho de banda de acceso a memoria; y los cores de cada procesador, en los que se debe indicar la frecuencia, la potencia estática, la potencia dinámica, etc.

- *Estudiar el comportamiento del planificador*: Una vez tenemos todas las partes definidas y la simulación está lista para comenzar, debemos analizar de forma teórica los resultados que esperamos obtener ya que si no, no seríamos capaces de sacar conclusiones a partir de los valores obtenidos.

  En este caso se hace uso de la planificación Backfill, además de algún algoritmo clásico, como es FIFO por similitud, o Random. Para los experimentos que se realizan con una cantidad de tareas moderada y una plataforma con pocos recursos, se realiza el análisis teórico completo y así nos aseguramos que el funcionamiento de la planificación es completamente correcto, y los valores obtenidos son los esperados. Sin embargo cuando la prueba crece en dimensiones, simplemente se analizan los resultados ya que resulta inviable realizar el estudio completo.

- *Realizar el experimento*: Se realiza el experimento lanzando el comando por terminal de la forma ïrmasim options.json". En el fichero options.json se indica el fichero que

describe la plataforma y el que detalla el conjunto de jobs, además de otros parámetros fundamentales.

- *Comprobar los resultados*: Una vez finalizado el experimento, se comprueban los resultados obtenidos para verificar si el planificador se ha comportado de la forma adecuada o ha ocurrido algo inesperado. La ejecución de la prueba y los resultados se pueden observar tanto en la salida de la terminal, la cual va mostrando como se realiza la simulación y el tiempo que ha durado, como en los siguientes ficheros de salida:

  - *info.log*: Recoge los diferentes eventos ocurridos durante la simulación, como puede ser por ejemplo el momento de llegada de los jobs a la plataforma.
  - *jobs.log*: Recoge información acerca de todos los jobs ejecutados en la plataforma. Detalla información útil como el tiempo de llegada, el tiempo de finalización o los cores empleados.
  - *makespans.log*: Muestra la latencia de finalización del conjunto de jobs para obtenerla de una forma ágil.
  - *speedup.log*: Recoge valores que a priori no se les puede dar una interpretación por si solos, pero que son útiles para la creación de estadísticas y gráficas.
  - *statistics.json*: Muestra las métricas fundamentales de la simulación como pueden ser la energía consumida, el EDP (Energy Delayed Product) [23] o el makespan.

  Con todos estos datos ya tenemos las herramientas suficientes para valorar si el experimento ha resultado satisfactorio.

Una vez concluida la prueba, se puede valorar si esta ha sido un éxito o un fracaso en función de los resultados obtenidos. Si los valores que devuelve la herramienta coinciden con los que teníamos previstos teóricamente, podemos concluir que el algoritmo se comporta correctamente. En ocasiones puede ocurrir que los resultados parecen contradictorios, pero analizándolos detenidamente nos damos cuenta de que era una situación que no habíamos contemplado y podría ocurrir ocasionalmente. También puede ocurrir que los resultados sean muy dispares y no consigamos encontrarles una respuesta lógica. En ese caso nos aseguramos de que los valores obtenidos se repiten continuadamente y en consecuencia, acotamos las deficiencias de este tipo de planificación.

Para asegurarnos que los resultados no son obtenidos de forma azarosa, cada experimento es repetido 20 veces, de acuerdo a un script. Con esto nos aseguramos que los resultados obtenidos son fiables, y en base a ellos podemos asegurar las premisas planteadas con certeza.

Todas las pruebas han sido realizadas en un ordenador portátil, concretamente el modelo MSI Modern 15 A11SB-011ES que cuenta con un procesador Intel Core i7-1165G7, 16 GB de memoria RAM, 1 TB de almacenamiento de estado sólido y una tarjeta gráfica Nvidia MX450. El sistema operativo empleado es Ubuntu en su versión 21.04, con el kernel de Linux versión 5.11.0-49-generic.

4.2. Experimentos de planificación

En este apartado se realiza la comprobación efectiva del rendimiento que obtiene la nueva implementación, descrita en el Capítulo 3, y se realiza una comparativa con diversos algoritmos clásicos. Los experimentos se detallan para que puedan ser replicados y el orden de aparición en el documento es en complejidad creciente.

4.2.1. Ejemplo introductorio

Para familiarizarse con el comportamiento del algoritmo, en primer lugar se va a realizar un pequeño experimento en el que se disponen cuatro jobs en una plataforma de dos cores. Este resulta muy útil para comprender cómo funciona el algoritmo de forma teórica ya que la planificación se puede realizar a mano.

El conjunto de tareas es el siguiente:

- Job 1: Subtime = 2, Recursos = 1, EET = 10
- Job 2: Subtime = 2.5, Recursos = 2, EET = 12
- Job 3: Subtime = 2.7, Recursos = 1, EET = 8
- Job 4: Subtime = 2.8, Recursos = 1, EET = 5

El planteamiento se puede seguir en la figura 4.1. En ella podemos observar en el eje X el transcurso del tiempo, y en el eje Y los diferentes jobs. Las flechas encima del eje X indican los tiempos de llegada de los trabajos a la plataforma. Respecto a los jobs, la zona rayada representa el tiempo bloqueados y la zona lisa el tiempo en ejecución.

En el segundo 2 de la simulación se recibe el primer job que, como tiene todo el sistema libre, entra a ejecutar sin problema. En el segundo 2.5 llega el segundo job que como necesita dos cores, y solo hay uno disponible, es bloqueado. En el segundo 2.7 llega el tercer job y se procede a realizar la planificación. Se intenta planificar el segundo job, pero sigue sin tener los recursos disponibles y, al quedar más trabajos pendientes, se calcula el backfill gap. En este caso el valor del backfill gap es 12, ya que el job 1 se planificó en el segundo 2 y tiene un *EET* de 10. El algoritmo busca si puede ejecutar algún job en Backfill, y el tercer job reúne las condiciones adecuadas, ya que necesita 1 solo core y su tiempo de finalización estimado es 10.7, por lo que puede ejecutarse antes que el Job 2 sin molestarle. Para el segundo 2.8 se recibe el cuarto job y ni el segundo, ni este nuevo pueden ejecutar ya que el sistema está completo.

En el segundo 3 de la simulación, el primer job finaliza su ejecución por lo que libera sus recursos y se trata de planificar los jobs pendientes. El segundo job sigue sin poder ejecutar

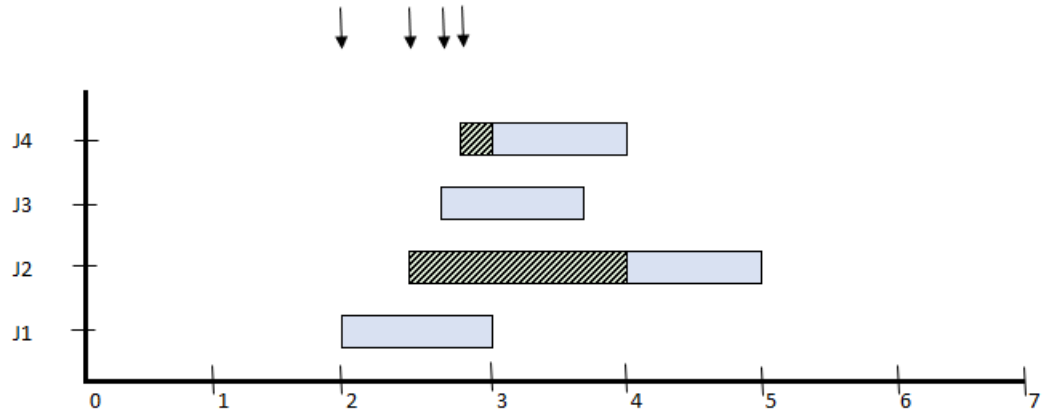

Figura 4.1: Análisis de Backfill con buena estimación de EET

ya que hay un core libre pero necesita dos, por lo que se calcula el backfill gap nuevamente y, ahora en vez de ser 12 el resultado, es 10.7 (reduciéndose así la ventana de backfill). Se busca entre los trabajos pendientes si existe alguno que pueda ejecutar y el cuarto job reúne las condiciones necesarias, ya que tiene como tiempo estimado de finalización 8 (nos encontramos en el segundo 3 y su *EET* es 5), por lo que adelanta su ejecución. En el segundo 3.7 el tercer job finaliza su ejecución y se intenta planificar la segunda tarea, pero esta sigue sin disponer de todos los recursos. Por último, en el segundo 4 el cuarto job finaliza su ejecución y el job bloqueado ya tiene todos los recursos necesarios disponibles por lo que entra a ejecutar y finaliza en el segundo 5.

Como podemos observar, esta simulación nos indicaría un resultado de 5 en el makespan, y debido a los *EET* estimados se ha producido Backfill en dos ocasiones.

Sin embargo podemos lograr una ejecución diferente simplemente modificando uno de los valores de los jobs. El nuevo conjunto de jobs es:

- Job 1: Subtime = 2, Recursos = 1, EET = 10
- Job 2: Subtime = 2.5, Recursos = 2, EET = 12
- Job 3: Subtime = 2.7, Recursos = 1, EET = 8
- Job 4: Subtime = 2.8, Recursos = 1, EET = 8

El único valor modificado es el *EET* del Job 4, que ha pasado de 5 a 8.

Si realizamos el estudio al igual que antes, todo se ejecuta igual hasta el segundo 3. En ese momento el primer job finaliza su ejecución por lo que libera sus recursos y se comprueba si existe la posibilidad de realizar *Backfill*. El segundo job sigue sin poder ejecutar ya que hay un core libre pero necesita dos, por lo que se calcula el backfill gap nuevamente, que

es de 10.7. Ahora, el Job 4 no puede adelantar su ejecución ya que su tiempo estimado de finalización es de 11 (estamos en el segundo 3 de simulación y su *EET* es de 8). En el segundo 3.7 el Job 3 finaliza su ejecución y de esta forma el Job 2 que estaba bloqueado en la cabeza entra a ejecutar al disponer de todos los recursos necesarios. Cuando este finaliza su ejecución en el segundo 4.7, el Job 4 entra a ejecutar y finaliza en el segundo 5.7.

Este planteamiento, al igual que se hizo con el anterior, se puede seguir en la figura 4.2. En ella podemos observar en el eje X el transcurso del tiempo, y en el eje Y los diferentes jobs. Las flechas encima del eje X indican los tiempos de llegada de los trabajos a la plataforma. Respecto a los jobs, la zona rayada representa el tiempo bloqueados y la zona lisa el tiempo en ejecución.

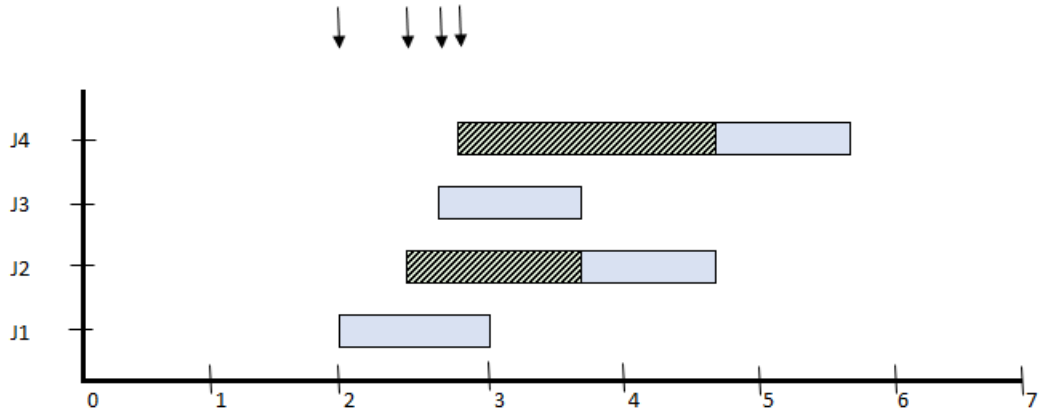

Figura 4.2: Análisis de Backfill con mala estimación de EET

Esta simple situación nos hace reflexionar acerca de la importancia que tiene la precisión a la hora de elegir el *EET* aportado por el usuario al lanzar el trabajo. En el primer caso, en el que el valor se aproxima más a la ejecución real, los jobs son más susceptibles de ser seleccionados para adelantar su ejecución lo que hace que el tiempo se optimice en mayor medida. Sin embargo, en el segundo caso el valor está sobredimensionado, lo que induce al sistema a pensar que el trabajo va a estar corriendo más tiempo del que en realidad necesita. Esto hace que no se pueda maximizar la utilización del sistema y por tanto otro tipo de algoritmos darían mejor resultado.

Si hacemos el mismo análisis empleando una política FIFO, observamos lo siguiente. En el segundo 2 llega el Job 1 y entra a ejecutar puesto que el sistema al completo está libre. A continuación, en el segundo 2.5 llega el Job 2 y como no tiene recursos suficientes disponibles se bloquea en la cabeza de la cola. En los segundos 2.7 y 2.8 llegan los Jobs 3 y 4 respectivamente, que también son bloqueados. El Job 1 finaliza su ejecución en el segundo 3, y el Job 2 entra a ejecutar ya que tiene los dos cores necesarios libres. Para el segundo 4 el Job 2 finaliza su ejecución y tanto el Job 3 como el Job 4 entran a ejecutar puesto que necesitan un core cada uno, de los dos disponibles y solapan su ejecución.

Por el momento vemos como no existe un claro ganador entre ambos algoritmo ya que en

unas ocasiones el beneficiado es *Backfill* y en otras *FIFO*, siempre teniendo en cuenta el valor de *EET* aportado por el usuario. En este ejemplo el makespan obtenido tanto con *FIFO* como con *Backfill* con buenas estimaciones de *EET* es el mismo, pero como veremos más adelante, no debemos fijarnos únicamente en la latencia como medida del rendimiento ya que hay otras métricas que nos pueden resultar incluso más interesantes.

4.2.2. Experimentos de gran escala

Una vez tenemos claro como debería funcionar el algoritmo y que la implementación realizada es correcta, pasamos a comprobar su funcionamiento en entornos de una mayor entidad.

En este experimento, la plataforma sobre la que se va a realizar la prueba está formada por un cluster con 20 nodos, y cada uno de esos nodos dispondrá de un procesador con 8 cores. Esto hace que el agregado de cores en el sistema sea de 160. Todos los cores tienen las mismas características en cuanto a potencia, frecuencia, etc.

Esta traza ya se asemeja en algún modo a un pequeño datacenter que se podría tener por ejemplo en alguna empresa o entidad de mediano tamaño, ya que se dispone de un cierto nivel de capacidad de cómputo.

En cuanto al conjunto de jobs, se dispone de 40 trabajos que pueden ser de 6 tipos en función de la necesidad de recursos que tengan. Las características de cada tipo de Job se recogen en el Cuadro 4.1. La traza se compone de forma que los jobs se añaden al fichero jobs.json de forma ordenada y cuando se llega al Job con necesidad de 128 cores, el siguiente vuelve a ser el de 4. Esto se repite hasta completar los 40 Jobs.

Tipo de Jobs	**Subtime**	**Cores necesarios**	**EET**	**Perfil**
Job 0	0	4	7	Clase 1
Job 1	0	8	7	Clase 1
Job 2	0	16	7	Clase 1
Job 3	0	32	7	Clase 1
Job 4	0	64	14	Clase 2
Job 5	0	128	14	Clase 2

Cuadro 4.1: Configuración de Jobs para el primer experimento.

Como podemos ver en el Cuadro 4.1, el subtime de todos los Jobs es 0 por lo que al comienzo de la simulación ya se conoce el conjunto al completo y esto favorece que en caso de que se pueda, *Backfill* aparezca más frecuentemente. Además, se observa que existen 2 tipos de Jobs (Clase 1 y Clase 2). Esto es así ya que los Jobs que necesitan más cores tienden a ser más

demandantes por lo que tardarán más en ejecutar. Los Jobs de la Clase 1 necesitan ejecutar 5000000 de operaciones con un IPC de 1, por lo que tardan 5 segundos, mientras que los Jobs de la clase 2 necesitan ejecutar 10000000 de instrucciones con un IPC de 1 también, por lo que tardan el doble, es decir 10 s. Al igual que sucede con las clases, el valor de *EET* aproximado también se duplica en el caso de la segunda clase para que sea más justo.

Con todos estos datos en los ficheros de platform.json y jobs.json, procedemos a realizar el experimento, ejecutando cada prueba varias veces para asegurarnos que los resultados obtenidos son correctos.

En primer lugar, realizamos la simulación empleando el planificador *Backfill*. En terminal se observa que se produce adelantamiento de Jobs, pero que el final de la simulación viene marcado por los Jobs más demandantes y esto hace que el final acabe comportándose prácticamente como *FIFO*, ya que los Jobs que necesitan 128 cores van a ejecutar solos en el sistema.

A continuación realizamos la simulación con los demás algoritmos, que son *FIFO*, *Random* y *SJF*.

Los resultados obtenidos en cuanto a makespan los podemos ver en la Gráfica 4.3. En ella observamos que tanto *Backfill*, como *FIFO* y *Random* se comportan de igual en términos de latencia, ya que como se indicaba antes, esta viene determinada por los trabajos más demandantes, y aunque los Jobs que necesitan menos recursos puedan ejecutar en medio, los grandes nos van a limitar en este término. Además vemos como el algoritmo *SJF* produce un makespan superior por lo que el sistema está desaprovechado una mayor parte del tiempo.

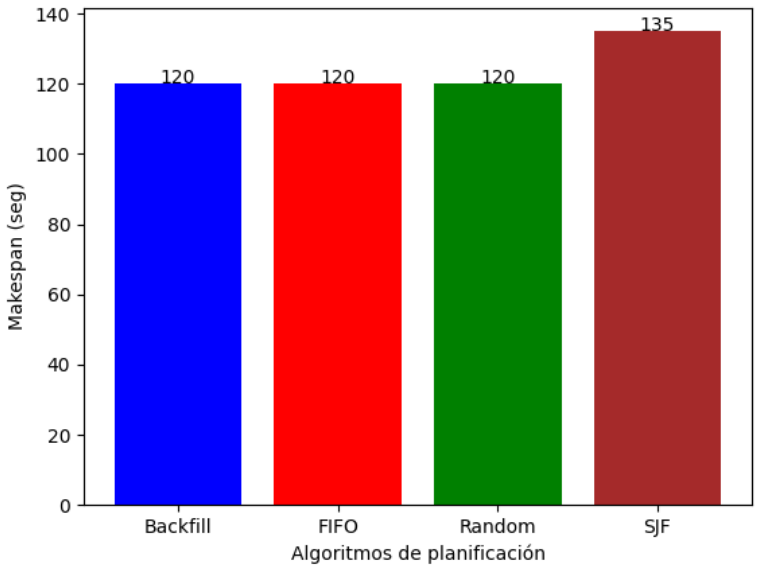

Figura 4.3: Makespan obtenido en el experimento con 40 Jobs

Sin embargo, si analizamos otra métrica también muy empleada como es el AWT (Average

Waiting Time), vemos como los resultados difieren. Esta medida nos indica el tiempo medio que deben esperar los Jobs desde que son cargados en el sistema hasta que empiezan a ejecutar.

Los valores aportados por la herramienta los podemos visualizar en la Gráfica 4.4. En esta ocasión vemos como el resultado para *SJF* es más favorable ya que este algoritmo tiene como objetivo dar cabida a los jobs menos demandantes primero, y no es necesario esperar a que los pesados ejecuten antes. Como hay más trabajos ligeros que pesados en la plataforma, por eso el resultado es tan positivo. Además, conviene aclarar que *SJF* es un algoritmo teórico y que no puede ser implementado en la realidad ya que en el momento en que se lanza un trabajo, no se sabe cuál va a ser su tiempo de ejecución.

Sin embargo vemos como ahora sí que existe diferencia entre *Backfill* y *FIFO*. Esto se debe a que en *FIFO*, todos los jobs deben esperar a que el de 128 cores finalice para poder ejecutar, mientras que en *Backfill*, en la mayoría de ocasiones los trabajos menos demandantes pueden adelantar su ejecución.

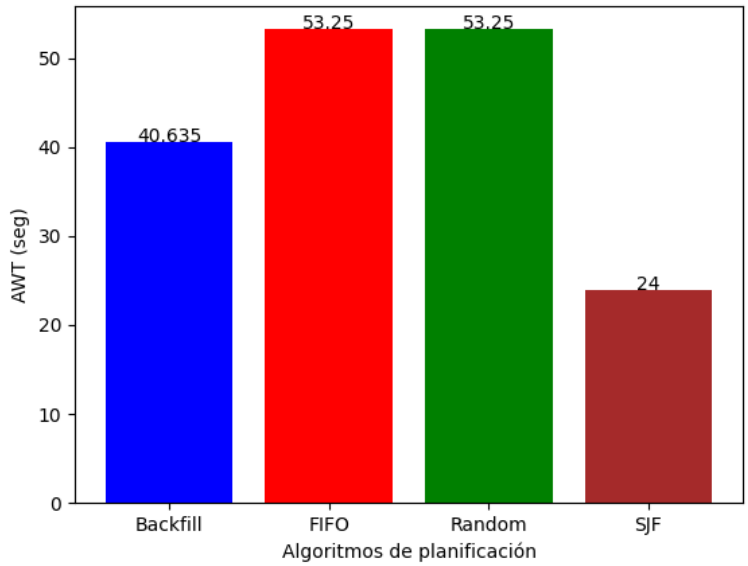

Figura 4.4: AWT obtenido en el experimento con 40 Jobs

En base a estos resultados, podemos ver cómo el algoritmo *Backfill* mejora o iguala a *FIFO* en rendimiento, que es el principal planificador a comparar, dadas sus similitudes, pero no es el mejor en todos los aspectos ya que por ejemplo *SJF*, mejora algunas métricas y empeora otras.

Para el siguiente experimento, la plataforma que se va a emplear es similar a la anterior, pero con una mayor capacidad de cómputo. En este caso se dispone de 25 nodos, los cuáles disponen de 1 procesador con 8 cores cada uno. Esto hace un total agregado de 200 cores.

En cuanto al conjunto de Jobs, se va a duplicar el número de trabajos a ejecutar. En el

anterior experimento se disponía de 40 y ahora vamos a emplear 80. Respecto a la naturaleza de los jobs, vamos a emplear 7 tipos de trabajos, los cuáles se recogen en el Cuadro 4.2.

Tipo de Jobs	**Subtime**	**Cores necesarios**	**EET**	**Perfil**
Job 0	0/30	4	7	Clase 1
Job 1	0/30	8	7	Clase 1
Job 2	0/30	16	7	Clase 1
Job 3	0/30	20	7	Clase 1
Job 4	0/30	32	7	Clase 1
Job 5	0/30	64	14	Clase 2
Job 6	0/30	128	14	Clase 2

Cuadro 4.2: Configuración de Jobs para el segundo experimento.

Los primeros 5 tipos van a ser considerados como ligeros y los 2 últimos como pesados por lo que, en principio, esto va a favorecer a que ocurra más veces el adelantamiento de trabajos. Además, se prevé que el algoritmo *SJF* va a tener un buen rendimiento en este caso ya que existen más trabajos ligeros que antes, y estos van a ser planificados en primer lugar.

Además de duplicar el número de Jobs y añadir un tipo más, se modifica el subtime de ciertos Jobs para que no existan todos ya en la plataforma al comienzo de la simulación. En este caso, los primeros 40 van a tener un subtime de 0 y los 40 siguientes van a tener un subtime de 30. Con esto se consigue que no siempre se haga el *Backfill* con los trabajos de 4 y 8 recursos mientras en el sistema están corriendo los demás trabajos ligeros. Así vemos como se mezcla la ejecución de trabajos demandantes y ligeros.

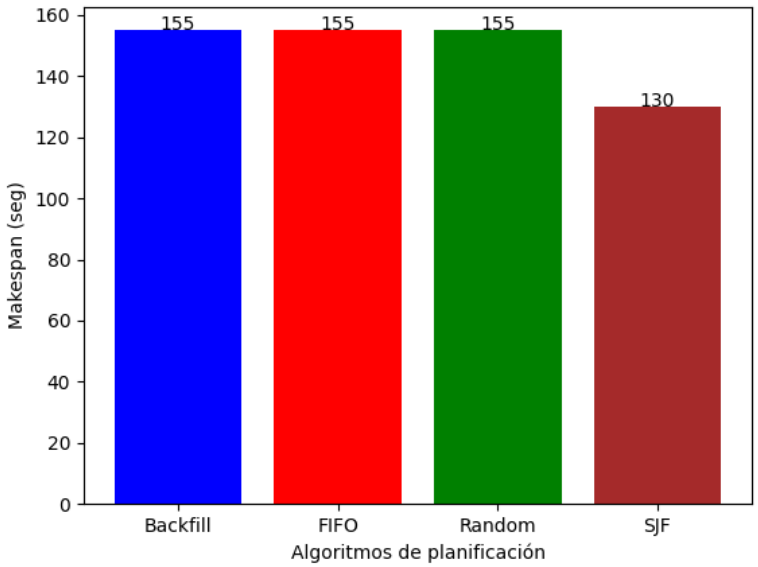

Figura 4.5: Makespan obtenido en el experimento con 80 Jobs (Menos demandantes)

En primer lugar, al igual que antes, se obtienen las medidas de latencia de los 4 algoritmos

que estamos analizando, *Backfill*, *FIFO*, *Random* y *SJF*. Los resultados obtenidos se pueden observar en la Gráfica 4.5.

Como podemos apreciar, los valores obtenidos tanto en *Backfill*, como en *FIFO* y en *Random* son los mismos, al igual que ocurría en el primer experimento, pero esta vez el algoritmo *SJF* consigue reducir la latencia, obteniendo un mejor rendimiento.

Esto lo podemos explicar desde el punto de vista de que, precisamente este algoritmo trata de planificar en primer lugar los trabajos menos demandantes y al estar tan desbalanceada la diferencia entre ambos tipos de Jobs, los más ligeros no deben esperar en ningún momento a los pesados.

Además observamos que respecto a *FIFO*, de momento *Backfill* nunca ha obtenido un peor resultado en ninguna métrica.

Sin embargo, si analizamos otra métrica como es el AWT, vemos que los resultados varían respecto al anterior experimento. Los valores obtenidos tras las pruebas se representan en la Gráfica 4.6.

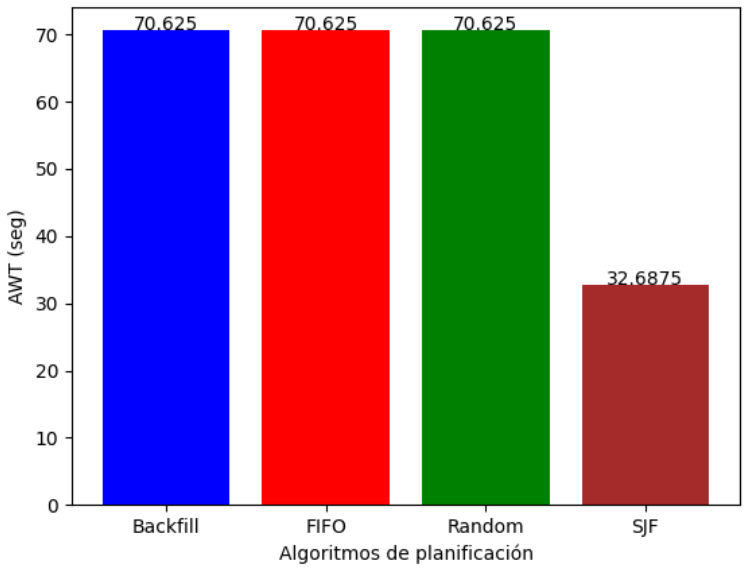

Figura 4.6: AWT obtenido en el experimento con 80 Jobs (Menos demandantes)

En este caso, al igual que ocurre con la latencia, los dos principales algoritmos a comparar (*Backfill* y *FIFO*), además de *Random*, obtienen el mismo resultado. Justo para este tipo de traza resulta indiferente emplear un tipo de planificador u otro ya que los valores obtenidos en las métricas analizadas es el mismo. Esto se debe a que, a pesar de que los Jobs puedan adelantar la ejecución, al final, la mezcla entre los distintos trabajos hace que la ejecución se solape y la ganancia se pierda.

Por último se va a realizar un experimento similar al anterior pero cambiando la naturaleza

de los jobs. Ahora en vez de disponer de una mayor cantidad de trabajos menos demandantes, se van a emplear más del segundo tipo.

Los valores a utilizar se recogen en el Cuadro 4.3

Tipo de Jobs	Subtime	Cores necesarios	EET	Perfil
Job 0	0/30	4	7	Clase 1
Job 1	0/30	8	7	Clase 1
Job 2	0/30	16	7	Clase 1
Job 3	0/30	32	7	Clase 1
Job 4	0/30	64	14	Clase 2
Job 5	0/30	96	14	Clase 2
Job 6	0/30	128	14	Clase 2

Cuadro 4.3: Configuración de Jobs para el tercer experimento.

En este caso se sustituye el trabajo que habíamos incorporado, que necesitaba 20 recursos, por uno que necesita 96. Ahora existe un mayor equilibrio entre trabajos demandantes y no demandantes, por lo que ahora el rendimiento obtenido por *SJF* no será tan bueno.

La plataforma que vamos a utilizar ahora es la misma que en el anterior experimento. Se trata de un cluster con 25 nodos, y cada uno de ellos dispone de un procesador con 8 cores, lo que hace un total agregado de 200.

Al igual que en las anteriores pruebas, vamos a comenzar analizando la latencia.

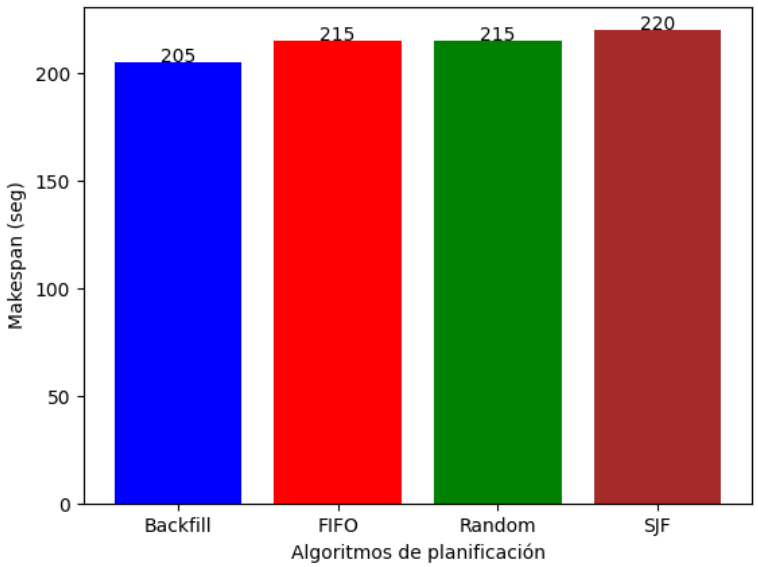

Figura 4.7: Makespan obtenido en el experimento con 80 Jobs (Más demandantes)

Como vemos en la Gráfica 4.7, esta vez sí que es *Backfill* el que obtiene un mejor valor en

cuanto a Makespan, mientras que en los anteriores casos había estado empatado con los otros algoritmos.

Teniendo esto en cuenta podemos determinar que, en cuanto a latencia, *Backfill* se va comportar como mínimo al igual que *FIFO* ya que aprovecha mejor el tiempo en la plataforma, por lo que resulta más recomendable utilizar el algoritmo que hemos implementado.

Con *SJF* al igual que en el primer experimento, la latencia obtenida es peor debido a la presencia de trabajos más demandantes que no permiten solapar tanto la ejecución como en el sefundo experimento.

A continuación realizamos el análisis del AWT y obtenemos los siguientes resultados, reflejados en la Gráfica 4.8.

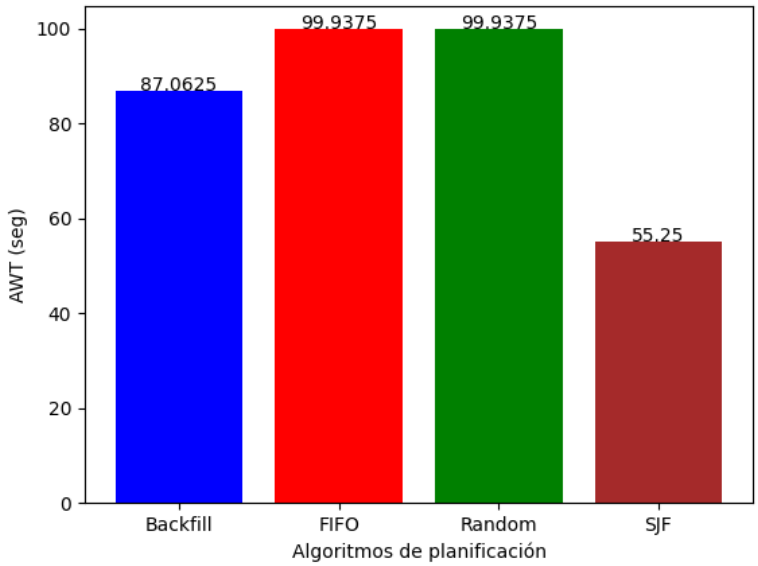

Figura 4.8: AWT obtenido en el experimento con 80 Jobs (Más demandantes)

En este caso, se sigue el mismo patrón que en el primer experimento en el que *FIFO* y *Random* se comportan igual pero *Backfill* consigue reducir el valor en torno a un 12 %.

Como siempre, el mejor resultado viene dado por el algoritmo *SJF*, que resulta muy eficiente para esta métrica.

La verdadera comparación en estos experimentos es la que se realiza entre la planificación *Backfill* y *FIFO*, ya que *SJF* tiene como objetivo unos parámetros de planificación diferentes, además de tratarse de un algoritmo teórico que no se puede implementar con precisión en la práctica ya que requiere conocer anticipadamente el tiempo de ejecución de forma precisa. Este tiempo, además, puede variar notablemente si los trabajos caen en nodos compartidos con otros, lo que puede producir contención tanto en el acceso a memoria como en las comunicaciones, aumentando el tiempo de ejecución del job, por encima de lo previsto. Por

todo esto, esta comparación no resulta tan útil, pero está bien para hacerse una idea de los valores que se pueden obtener con uno u otro, a pesar de que la forma de funcionar es diferente.

En cuanto a *Backfill* y *FIFO*, vemos como los resultados obtenidos han sido siempre iguales o mejores utilizando *Backfill*, ya que tarda lo mismo o menos en realizar la ejecución completa del conjunto de jobs, y además los jobs, por lo general, tienen un tiempo de espera bastante inferior desde que llegan hasta que son planificados.

Con estos resultados podemos confirmar que hemos conseguido el propósito fundamental de este proyecto que era implementar el algoritmo *Backfill* en el simulador *IRMaSim* y ver que, efectivamente consigue obtener un mejor rendimiento que *FIFO*, haciendo un uso más eficiente del tiempo de ejecución en la plataforma.

Capítulo 5

Conclusiones

En este capítulo final se desglosan los objetivos conseguidos con este proyecto y, a su vez, las posibles mejoras que se podrían implementar en el simulador IRMaSim, en concreto en lo referido a planificación de jobs, en el futuro.

5.1. Objetivos Conseguidos

Finalizado este trabajo, resulta relevante hacer una vista en retrospectiva de los objetivos iniciales descritos en la Sección 1.2. A continuación se hace un mapeo entre los logros conseguidos y los objetivos propuestos:

- *Adición de un nuevo algoritmo de planificación en el simulador*: La herramienta ha sido modificada con éxito para ser capaz de implementar una nueva política de planificación. Dada la naturaleza modular del código con el que está diseñado el simulador, se ha acoplado la nueva funcionalidad sin modificar en gran medida las capacidades ya había disponibles. Este avance dota al simulador de una mayor flexibilidad a la hora de seleccionar qué política se quiere emplear a la hora de planificar tareas. Además, se ha observado la importancia de los diferentes parámetros de entrada de la herramienta, en especial el *EET*. En función de cómo de buena sea la aproximación aportada por el usuario, el rendimiento del algoritmo aumenta o decrece. Cuanto más se acerque el valor a la ejecución real, mejores resultados se obtendrán, mientras que si las estimaciones son malas, la planificación tenderá a comportarse como *FIFO*, sin producirse apenas adelantamientos entre los Jobs.

  También se han conseguido modificar algunas áreas de la herramienta para hacerla más flexible y que futuros cambios puedan ser realizados de forma más sencilla. Esto hace que IRMaSim sea más fácil de entender desde el punto de vista interno y, sabiendo lo que se quiere implementar, sea más fácil encontrar los puntos a modificar.

- *Comparativa con algoritmos clásicos*: En base a los resultados obtenidos en el Capítulo 4 podemos obtener diversas conclusiones. En primer lugar, como vimos en el Capítulo 2, el problema de planificación es muy complejo y por tanto, este algoritmo no es el ideal para todo tipo de ocasiones, si no que en función del objetivo que queramos seguir, puede haber alguno mejor que sea específico en optimizar esa métrica. Pero se puede observar que comparado con otros algoritmos de su misma naturaleza como puede ser *FIFO*, *Backfill* sale ganando en la gran mayoría de ocasiones, tanto en latencia como sobre todo en AWT. La utilización que se consigue del sistema es mayor y los Jobs tienen que esperar bastante tiempo menos de media desde que entran al sistema hasta que comienzan a ejecutar. Además, como podemos observar en las diferentes gráficas, *Backfill* se comporta de una forma muy equilibrada ya que, a pesar de no conseguir los mejores valores en todo momento, siempre se mantiene en un término medio. Otro tipo de algoritmos tienden a optimizar mucho una métrica, pero para las demás no son tan adecuados. En este sentido, este algoritmo es muy versátil y puede emplearse ampliamente en distinto tipo de ejecuciones ya que somos conscientes que los resultados van a ser buenos, aunque no siempre estrictamente los mejores.

Estos items consiguen el propósito fundamental del desarrollo de este trabajo, el cual era implementar un nuevo algoritmo de planificación en el simulador IRMaSim, y contrastar su rendimiento con otro tipo de algoritmos, consiguiendo resultados con una tasa de error muy baja y grandes speedups en cuanto a velocidad de ejecución.

5.2. Trabajos Futuros

Durante el desarrollo de este proyecto han surgido nuevas vías de mejora para la herramienta pero que por falta de tiempo no han podido ser llevadas a cabo. Estas se dejan abiertas a una futura implementación y son listadas a continuación:

- *Red de interconexión*: Actualmente el simulador está diseñado y funciona correctamente con entornos heterogéneos en los que se puede introducir una alta carga de trabajo. Los jobs son asignados a los recursos (cores con su respectiva memoria), y simulada su ejecución. Una vez finalizan liberan esos recursos y otro trabajo puede hacer uso de ellos. Pero la realidad es que el aumento del desarrollo y las mejoras producidas en el área de la informática actualmente y en los próximos años, hace que se tengan cada vez trabajos de mayor complejidad y crecientes necesidades. En centros de procesado de datos de una cierta dimensión no simplemente se asignan las tareas a los cores de la CPU, y estas ejecutan independientemente, si no que existe comunicación entre los trabajos. Una posible propuesta sería adaptar esta situación real a la herramienta para que las tareas puedan ser planificadas en diferentes nodos y estos se comuniquen entre sí a través de una red de interconexión. De esta forma se dispondría de jobs de tipo MPI, que no realizan el intercambio de información a través de la memoria del nodo.

- *Uso de hardware dedicado*: La herramienta permite modelar clusters heterogéneos haciendo uso de principalmente dos tipos de componentes: CPUs y Memoria. Estos pueden ser de diferente tipo, y cada uno disponer de sus propias características, lo que modifica la simulación y hace que los resultados sean más fieles a la realidad. Sin embargo, realmente existen más piezas que conforman un entorno computacional real, como pueden ser las GPUs o los procesadores vectoriales. Resultaría muy interesante modelar estos componentes ya que el rendimiento que consiguen frente a los elementos tradicionales es muy superior, y son empleados cada vez en mayor medida.

- *Implementación de diferentes algoritmos*: En este proyecto se ha conseguido implementar el algoritmo Backfill en la plataforma, y se han podido contrastar sus beneficios o desventajas en función de las estimaciones introducidas como *EET*. Para dotar de una mayor variedad a la planificación de tareas resulta necesario incrementar el número de algoritmos disponibles. El simulador cuenta actualmente con planificación FIFO, Backfill o HDeepRM haciendo uso de inteligencia artificial. Se podrían añadir tanto algoritmos clásicos, como SJF o Round Robin mencionados en el Capítulo 2, u optar por implementaciones más modernas que fijen su objetivo en tratar de minimizar algún aspecto en específico.

Bibliografía

[1] Adrián Herrera Arcila et al. Hdeeprm: Deep reinforcement learning for workload management in heterogeneous clusters. 2019.

[2] Thomas Sterling, Matthew Anderson, and Maciej Brodowicz. *High Performance Computing: Modern Systems and Practices*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2017.

[3] Luiz André Barroso, Urs Hölzle, and Parthasarathy Ranganathan. The datacenter as a computer: Designing warehouse-scale machines. *Synthesis Lectures on Computer Architecture*, 13(3):i–189, 2018.

[4] José Luis Bosque and L. P. Perez. Theoretical scalability analysis for heterogeneous clusters. In *4th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2004), April 19-22, 2004, Chicago, Illinois, USA*, pages 285–292. IEEE Computer Society, 2004.

[5] Jason Mars, Lingjia Tang, and Robert Hundt. Heterogeneity in "homogeneous" warehouse-scale computers: A performance opportunity. *IEEE Computer Architecture Letters*, 10(2):29–32, 2011.

[6] Esteban Stafford and José Luis Bosque. Performance and energy task migration model for heterogeneous clusters. *J. Supercomput.*, 77(9):10053–10064, 2021.

[7] Jeffrey D. Ullman. Np-complete scheduling problems. *Journal of Computer and System sciences*, 10(3):384–393, 1975.

[8] Poonam Singh, Maitreyee Dutta, and Naveen Aggarwal. A review of task scheduling based on meta-heuristics approach in cloud computing. *Knowledge and Information Systems*, 52(1):1–51, 2017.

[9] Di Zhang, Dong Dai, Youbiao He, Forrest Sheng Bao, and Bing Xie. Rlscheduler: An automated hpc batch job scheduler using reinforcement learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20. IEEE Press, 2020.

[10] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. Resource management with deep reinforcement learning. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, HotNets '16, page 50–56, New York, NY, USA, 2016. Association for Computing Machinery.

[11] A. Herrera, M. Ibáñez, E. Stafford, and J. L. Bosque. A simulator for intelligent workload managers in heterogeneous clusters. In *21st IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing, CCGrid, Melbourne, Australia, May 10-13, 2021*, pages 196–205, 2021.

[12] Dan Tsafrir, Yoav Etsion, and Dror G. Feitelson. Backfilling using system-generated predictions rather than user runtime estimates. 2007.

[13] Esteban Stafford and José Luis Bosque. Improving utilization of heterogeneous clusters. *J. Supercomput.*, 76(11):8787–8800, 2020.

[14] José Luis Bosque, Oscar David Robles, Pablo Toharia, and Luis Pastor. Evaluating scalability in heterogeneous systems. *J. Supercomput.*, 58(3):367–375, 2011.

[15] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. CreateSpace Independent Publishing Platform, North Charleston, SC, USA, 2018.

[16] Mario Ibáñez Bolado et al. Gestión de recursos en datacenters basado en deep reinforcement learning. 2020.

[17] Fengcun Li and Bo Hu. Deepjs: Job scheduling based on deep reinforcement learning in cloud data center. In *Proceedings of the 2019 4th international conference on big data and computing*, pages 48–53, 2019.

[18] Pamela Delgado, Diego Didona, Florin Dinu, and Willy Zwaenepoel. Kairos: Preemptive data center scheduling without runtime estimates. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 135–148, 2018.

[19] Léo Grange, Georges Da Costa, and Patricia Stolf. Green it scheduling for data center powered with renewable energy. *Future Generation Computer Systems*, 86:99–120, 2018.

[20] Yasir Noman Khalid, Muhammad Aleem, Radu Prodan, Muhammad Azhar Iqbal, and Muhammad Arshad Islam. E-osched: a load balancing scheduler for heterogeneous multicores. *The Journal of Supercomputing*, 74(10):5399–5431, 2018.

[21] Ruth Malan, Dana Bredemeyer, et al. Functional requirements and use cases. *Bredemeyer Consulting*, 2001.

[22] Lawrence Chung and Julio Cesar Sampaio do Prado Leite. On non-functional requirements in software engineering. In *Conceptual modeling: Foundations and applications*, pages 363–379. Springer, 2009.

[23] Emilio Castillo, Lluc Alvarez, Miquel Moretó, Marc Casas, Enrique Vallejo, José Luis Bosque, Ramón Beivide, and Mateo Valero. Architectural support for task dependence management with flexible software scheduling. In *IEEE International Symposium on High Performance Computer Architecture, HPCA 2018, Vienna, Austria, February 24-28, 2018*, pages 283–295. IEEE Computer Society, 2018.