Universidad Autónoma de Barcelona

Escuela de Ingeniería

Departamento de Arquitectura de Computadores

y Sistemas Operativos

Planificación de Diferentes Clases de Aplicaciones en Entornos No Dedicados Considerando Procesadores Multicore

Tesis doctoral presentada por **José R. García** para optar al grado de Doctor por la Universidad Autónoma de Barcelona, bajo la dirección del **Dr. Porfidio Hernández**

Barcelona, Mayo de 2010

Planificación de Diferentes Clases de Aplicaciones en Entornos No Dedicados Considerando Procesadores Multicore

Tesis doctoral presentada por José R. García para optar al grado de Doctor por la Universidad Autónoma de Barcelona. Trabajo realizado en el Departamento de Arquitectura de Computadores y Sistemas Operativos de la Escuela de Ingeniería de la Universidad Autónoma de Barcelona, dentro el Programa de Doctorado en Informática, opción A "Arquitectura de computadores y procesamiento paralelo" bajo la dirección del Dr. Porfidio Hernández Budé.

Barcelona, Mayo de 2010

Director

Dr. Porfidio Hernández

Es la libertad la esencia de la vida.
José Martí

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

A día de hoy es prácticamente imposible encontrar una gran institución que no disponga de un parque de ordenadores considerable, debido al alto nivel de informatización de la sociedad actual. El enorme potencial que representan estos miles de ordenadores atrae poderosamente la atención en los ámbitos científicos e industriales, generando opciones viables para su aprovechamiento. Las universidades, instituciones que históricamente se han mantenido a la vanguardia en la investigación e innovación científica, representan un caso especialmente bien posicionado a la hora de generar tanto los recursos informáticos como la necesidad de su uso.

El poder de cómputo presente en los laboratorios y aulas de estudio universitarias, agrupaciones naturales de recursos informáticos, crea grandes oportunidades para la computación paralela, animándonos a buscar opciones viables para su aprovechamiento. Como resultado de este interés, y con el propósito de aprovechar los períodos de inactividad de los nodos para ejecutar carga paralela, nuestro grupo ha creado un entorno de planificación, enfocado hacia los clusters no dedicados. La constante y rápida evolución de los componentes, tanto a nivel de la arquitectura de la CPU como del sistema operativo, así como de las aplicaciones ejecutadas, hace que tengamos que ir adaptando nuestras propuestas, que consideran tanto la simulación como las ejecuciones reales.

Nuestra propuesta consiste en crear una Máquina Virtual con una doble funcionalidad, ejecutar la carga local de usuario y aprovechar los períodos de inactividad de nodos a efectos de poder usarlos para ejecutar carga paralela. Tanto el tipo de las aplicaciones como las características del hardware del escenario objetivo, y en el momento actual ambas han evolucionado. Nuevos tipos de aplicaciones paralelas con requerimientos periódicos de CPU [112, 85, 116] son cada día más comunes en el mundo científico e industrial. Este tipo de aplicaciones pueden requerir un tiempo de retorno (*turnaround*) específico o una Calidad de Servicio (*Quality of Service*, ***QoS***) determinada,

haciendo más complejos los sistemas y modelos de predicción e imponiendo nuevas pautas en el desarrollo de los mismos.

Para nuestro caso particular, reviste especial importancia el conocimiento que poseemos de los usuarios locales, debido a que nuestro entorno está diseñado para trabajar en clusters no dedicados. Un usuario local puede estar visualizando un vídeo almacenado en su ordenador, lo cual implica necesidades de CPU periódicas y un mayor uso de memoria que los tipos de aplicaciones Best-effort habituales hasta la fecha. Estudios como [34] muestran diferentes aplicaciones empleadas por los usuarios locales en la actualidad, que necesitan la CPU de forma periódica y tienen mayores requerimientos de memoria principal y ancho de banda de red.

La aparición de nuevos tipos de aplicaciones, como vídeo bajo demanda, realidad virtual, aprendizaje a distancia y videoconferencias entre otras, se caracterizan por la necesidad de cumplir sus *deadlines* y por lo tanto presentan requerimientos periódicos de recursos. Este tipo de aplicaciones, donde la pérdida de deadlines no se considera un fallo severo, aunque ha de ser evitada en lo posible, han sido denominadas en la literatura aplicaciones *soft-real time* (SRT) periódicas.

Sin embargo, la interesante evolución de las necesidades de los usuarios no es el único punto a tener en cuenta. El crecimiento en la capacidad de cómputo de los procesadores en los últimos años se ha visto frenado a causa de las barreras físicas del espacio y la velocidad de las señales, obligando a los fabricantes de procesadores a explorar otras vías de crecimiento. Desde hace ya algún tiempo el paralelismo de las aplicaciones se ha convertido en una de las grandes apuestas [43]. Hoy en día los procesadores de dos núcleos son la mínima configuración que encontraremos en un ordenador, y las previsiones son que el número de núcleos por procesador continuará creciendo en los próximos años.

Los clusters no dedicados ofrecen un gran potencial de un uso, debido a que los recursos materiales ya están disponibles y el cálculo paralelo se realiza simultáneamente con el del usuario local. Imaginando el escenario actual en los clusters no dedicados, encontramos nuevas aplicaciones de escritorio y paralelas, así como plataformas hardware más potentes y complejas. En esta situación investigar el problema y realizar propuestas relacionadas con la planificación de los diferentes tipos de aplicaciones en clusters no dedicados, considerando las plataformas multi-core, supone un nuevo reto a asumir por los investigadores y conforma el núcleo de este trabajo.

1.1. Clusters no dedicados

Con objeto de satisfacer las necesidades de cómputo masivo existentes en la industria o la ciencia pura, la paralelización se ha convertido en una de las vías más estudiadas y aceptadas. En el fondo, el objetivo básico de esta estrategia consiste en mejorar el tiempo de ejecución de las aplicaciones para poder así afrontar problemas más complejos y aumentar el factor de escala. Cuando paralelizamos, es especialmente atractiva la idea de lograr una gran capacidad de cómputo a coste mínimo, siendo las redes de estaciones de trabajo (*Network of Workstations*, ***NOW***) no dedicadas una de las opciones. En nuestro enfoque, una NOW es un laboratorio docente universitario, un ambiente controlado donde conocemos y podemos estudiar a fondo las necesidades y costumbres de nuestros usuarios locales [11]. En las universidades existen todas las condiciones para este tipo de estudios, por un lado grandes redes de ordenadores pertenecientes a instituciones y por el otro, grupos científicos con necesidades de cómputo importantes.

Queremos destacar que el uso de los períodos de inactividad de los ordenadores pertenecientes a NOWs para ejecutar aplicaciones paralelas no es una utopía, pues en los escenarios antes descritos los ordenadores están ociosos entre el 60 y el 70 % del tiempo. Multitud de trabajos se han centrado en este enfoque desde diferentes perspectivas, que representan diferentes formas de utilizar esta capacidad de cómputo. A continuación mencionamos algunos de los enfoques centrados en clusters no dedicados, destacando que nuestro trabajo se enfoca en la *Planificación de Aplicaciones*:

- Máquinas Virtuales
- *Cluster Computing on The Fly* [74]: Técnica de cómputo oportunista.
- *Cluster-US* [87]: Basado en computadores hibernables, que utiliza los nodos para el para el cómputo paralelo en horarios en que nos se utilizan para otras tareas.
- *Grid* [81, 71]: Aprovechamiento de nodos ociosos para cómputo paralelo, en esta variante hay tanto clusters dedicados como no dedicados.
- **Planificación espacial y temporal de aplicaciones**: *Gang Scheduling* [38], Coplanificación dinámica [93], Coplanificación Dinámica [93] o sistemas como SLURM [60], diseñados para clusters Linux.

Nuestro campo de interés está centrado en los Multicomputadores, donde existen las opciones tipo *Massive Parallel Processors* (MPP) o *Commodity Of-The-Shelf* (COTS). La opción MPP proporciona una elevada capacidad de cómputo a un alto coste económico, ya que para su funcionamiento requiere de hardware y software específico. Por este motivo es cada vez más

común el uso de clusters COTS, debido principalmente a la disminución de los costes. Este tipo de entorno se construye a partir de componentes comerciales fácilmente accesibles y el nivel de acoplamiento que alcanzan es bastante menor que los MPP. Es también una opción válida proveer los clusters COTS de redes de conexión más rápidas, aunque hacerlo conlleva un aumento importante en el presupuesto.

Centramos nuestros estudios en los *clusters no dedicados*, este tipo de clusters está formados por recursos computacionales preexistentes, abaratándolos aún más con respecto a ordenadores tipo MPP. Es importante mencionar que en los clusters no dedicados podemos encontrar aplicaciones locales no controladas, y es precisamente la aparición de nuevos tipos de aplicaciones locales de tipo SRT una de las principales dificultades, en base al nuevo grado de complejidad que se introduce en el sistema. Esto unido a los nuevos requerimientos de las aplicaciones paralelas en cuanto al establecimiento de un turnaround determinado, constituyen el entorno donde se van a desarrollar las propuestas que se presentarán en este trabajo. La *planificación temporal* de aplicaciones Best-effort y SRT, tanto locales como paralelas, en entornos no dedicados de tal forma que no afectemos los niveles de respuesta del ordenador, necesarios para la comodidad de los usuarios locales, configurará uno de los objetivos fundamentales de las propuestas desarrolladas.

Son de suma importancia para nuestra finalidad estudios como [48, 32], enfocados tanto en el confort de los usuarios locales como en los recursos que emplean en sus aplicaciones. Aún cuando estos estudios se centran en aplicaciones tipo Best-effort, podemos complementarlos con estudios que caracterizan los nuevos tipos de aplicaciones locales SRT [35]. Nuestra finalidad es determinar la viabilidad de la ejecución de cómputo mediante un usuario paralelo sin que los niveles de respuesta del ordenador al usuario local se vean afectados.

Nuestro trabajo implica la coexistencia de la carga paralela con la presencia de usuarios locales(y sus aplicaciones) en los ordenadores. Esto condiciona nuestro problema de planificación a dos niveles, **planificación espacial** y **planificación temporal**. Es decir, necesitamos decidir dónde ejecutaremos nuestro cómputo paralelo y como planificaremos los recursos en los nodos compartidos, siendo este último tipo de planificación nuestro objetivo central. En otras partes de esta introducción presentaremos otras circunstancias que hacen aún más importante la necesidad de centrarnos en la planificación temporal, como lo es la presencia de procesadores multi-core.

1.2. Planificación en NOWs

Como ya hemos establecido anteriormente, la planificación de aplicaciones paralelas tiene dos áreas claramente diferenciadas, la *planificación espacial* y

la *planificación temporal*. De estas dos variantes, la primera es la que decide en qué conjunto de nodos se va a ejecutar una aplicación paralela y la segunda realiza la planificación temporal a corto plazo en cada nodo perteneciente al cluster. La Figura 1.1 muestra de forma general la problemática que abordaremos en esta sección.

Figura 1.1: Taxonomía General de la Planificación de Aplicaciones Paralelas

1.2.1. Planificación espacial

Las políticas de planificación espacial son las encargadas de decidir en qué nodos se ejecutan los trabajos y cómo se planificarán. En esta subsección abordaremos los temas relacionados con la selección de nodos y su posterior planificación. La selección de nodos podemos abordarla desde dos vertientes principales, la distribución de los nodos y la planificación de los trabajos.

1.2.1.1. Distribución de los nodos

Profundizando en la distribución de los nodos encontramos la necesidad de *particionar* y *seleccionar* los nodos para una correcta distribución. Entre las alternativas de **particionamiento** de nodos encontradas en la literatura, los más simples para su implementación son el estático y el variable, siendo este último una de las mejores opciones por el balance entre la simplicidad a la hora de implementarlo y las desventajas que presenta.

Por otro lado la **selección** intenta elegir los nodos donde ejecutaremos las aplicaciones paralelas de acuerdo a políticas de selección. La política de selección más simple, la *binaria*, considera que un nodo o bien está libre u ocupado. Este tipo de política no considera la posibilidad de que las aplicaciones puedan compartir nodos, ya que no tiene en cuenta el grado de ocupación de los nodos. Como ejemplo de política binaria encontramos las de tipo *buddy* [102, 73].

Las llamadas políticas de *selección discreta* [50, 40, 115] son aquellas dónde se considera un grado de multiprogramación (*Multi Programming Level*, ***MPL***)

mayor que 1. Lógicamente, al ser el MPL > 1, las políticas de este tipo han de combinar el espacio compartido con el tiempo compartido, es decir, trabajar espacial y temporalmente.

En caso de presentar un MPL > 1, es necesario de alguna forma poder estimar el grado de disponibilidad de los nodos para cómputo paralelo. Un nodo con cierto grado de ocupación podrá ejecutar una aplicación paralela o no de acuerdo a las necesidades de cómputo y memoria de esta. Dada la posibilidad de ejecutar aplicaciones paralelas en nodos donde se estén ejecutando aplicaciones locales, han de desarrollarse políticas que tengan en cuenta esta situación. Este tipo de políticas se conocen como de *selección continua*.

1.2.1.2. Planificación de Trabajos

La planificación de trabajos se centra en el ordenamiento de los trabajos en espera de ser ejecutados y la forma de seleccionarlos de la *cola de espera*. Debido a que nuestros esfuerzos se centran en la planificación temporal, no entraremos en grandes detalles en relación a la planificación espacial, explicando solo algunas de las políticas, para facilitar la comprensión de las empleadas durante la experimentación de este trabajo. Primero abordaremos las formas de ordenamiento de las colas de espera y luego la forma de seleccionar los trabajos a ejecutar.

Con el arribo al sistema de un nuevo trabajo paralelo que no se puede ejecutar en el momento, tenemos un incremento de la cola de espera del cluster. Algunas de las políticas de ordenamiento son: **FCFS** (*First Come First Served*, los trabajos son ejecutados en el orden en que llegan al sistema [114, 101]), **SJF** (*Shortest Job First*, los trabajos se ordenan de forma creciente en función del tiempo de ejecución estimado [6]) y **SNPF** (*Smallest Number of Processes First*, los trabajos se ordenan de acuerdo a la cantidad de procesadores que se solicitan [77]).

Una vez ordenada la cola de espera necesitamos *seleccionar los trabajos* que se encuentran en ella para su ejecución. Aunque elegir el primero parece la opción más justa, no siempre proporciona buenos resultados. Podría ocurrir que un trabajo con muchos requerimientos esté a la cabeza de la cola y frene innecesariamente los demás trabajos en espera. También ha de considerarse si además del orden existente en la cola de espera tendremos en cuenta el *estado actual* del cluster. El objetivo es lograr un equilibrio entre las métricas de utilización del sistema relacionadas con los usuarios y el rendimiento del cluster. Lo usual es intentar utilizar el conocimiento del estado del cluster para predecir el estado futuro y sacar ventajas de este conocimiento.

Entre las políticas más simples, que no necesitan información adicional excepto el conocimiento de la cola de espera podemos citar: *First Fit* y *Best Fit*. La primera de ellas, como su nombre indica, funciona buscando el primer

trabajo existente en la cola cuyos requerimientos de recursos sean menores o iguales que la disponibilidad de recursos del cluster [92, 6]. Empleando *Best Fit* el criterio de selección es que el trabajo elegido tenga los requerimientos de recursos más similares a la disponibilidad de recursos del cluster en el momento dado [114].

Entre las políticas que intentan obtener ventaja del *conocimiento del estado* del cluster podemos observar dos grupos fundamentales. Las que se basan en mantener una QoS durante la ejecución de la aplicación [23] y las que intentan adelantar trabajos utilizando técnicas de *Backfilling* [105, 53, 51]. La desventaja natural de este tipo de políticas reside en la necesidad de un **tiempo estimado** de ejecución de las aplicaciones. El problema representado por la imprecisión propia de las estimaciones de los usuarios [80] se ha intentado solucionar de utilizando diversas técnicas, ya sea mediante sistemas históricos [63, 107] o de modelos analíticos [59].

1.2.2. Planificación temporal

La necesidad de compartir los nodos entre los dos tipos de usuarios (local y paralelo) para lograr la planificación el clusters no dedicados, nos obliga a disponer de métodos para hacerlo de tal forma que no afectemos de forma perceptible la capacidad de respuesta del ordenador. Hemos de contemplar tanto las necesidades de los usuarios locales que brindan sus ordenadores, y encontrar formas de favorecer el progreso de las aplicaciones paralelas sin afectarlos. Debido a que los nodos en un cluster no dedicado son compartidos, han de mantenerse los niveles de respuesta del ordenador necesarios para el usuario local.

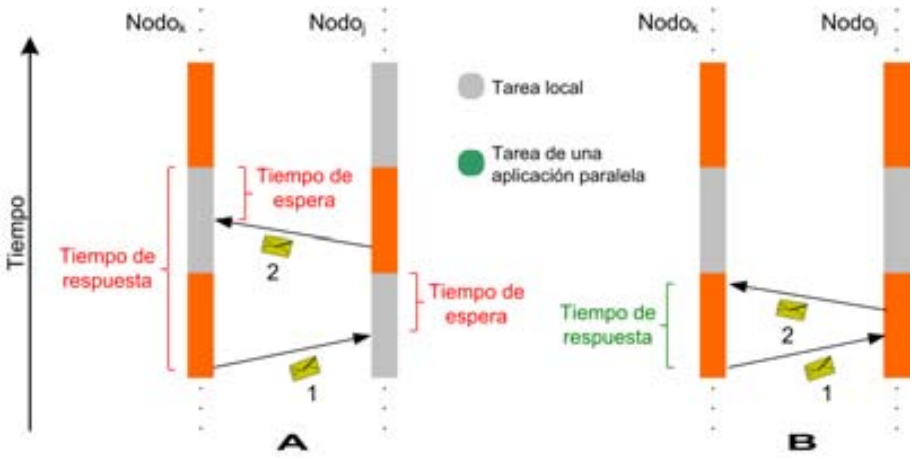

Figura 1.2: Efecto producido por la comunicación sobre la ejecución.

La forma de estimular el progreso de las aplicaciones paralelas se basa en su propia naturaleza. Podemos decir que las aplicaciones paralelas son procesos cooperantes, lo cual implica el intercambio de mensajes en mayor o menor medida. La Figura 1.2 muestra dos posibles casos de comunicación entre dos procesos cooperantes. En el caso A vemos claramente como el hecho de que dichos procesos cooperantes no dispongan de la CPU de forma simultánea provoca dos tiempos de espera adicionales. En cambio en el caso B los dos procesos cooperantes se planifican al mismo tiempo lo cual aumenta las probabilidades de disminuir los tiempos de espera provocados por la comunicación.

1.2.2.1. Coplanificación tradicional

La coplanificación, nombre recibido por las técnicas que intentan estimular la planificación simultánea de procesos cooperantes, fue introducida por [83] y ha sido abordada en múltiples estudios [45, 95, 13, 94, 38]. En su estudio embrionario, John Ousterhout propone un algoritmo conocido como **Algoritmo de la Matriz de Ousterhout** basado en la analogía existente entre la gestión de memoria en un entorno monoprocesador multiprogramado y la gestión de los procesadores en un entorno multiprocesador y multiprogramado. En un entorno monoprocesador y multiprogramado obtenemos una clara ventaja al tener todas las páginas del *working set* de una aplicación en memoria a la vez cuando se ejecuta. El estudio de Ousterhout mostró que el rendimiento de las aplicaciones paralelas se ve seriamente afectado si las mismas no reciben suficientes procesadores y las tareas pertenecientes a las mismas no son planificadas a la vez. Esto se debe a que los requerimientos de comunicación y sincronización existentes entre los procesos cooperantes de una aplicación paralela, pueden afectar ralentizando su ejecución debido a las esperas provocadas por la no planificación simultánea de sus procesos cooperantes.

La Figura 1.3 muestra un ejemplo de aplicación del Algoritmo de la Matriz de Ousterhout. La matriz está formada en el eje de las ordenadas por los procesadores ($j = 1 - n$) y en el eje de las abscisas por el número de máquinas virtuales en ejecución ($i = 1 - k$). Las máquinas virtuales tienen una potencia de cálculo igual a Pot_j/n donde Pot_j es la potencia de cálculo del procesador j y n es el grado de MPL de la máquina virtual. Cada columna contiene los procesos asignados a cada procesador Jp^k y cada fila los trabajos que serán ejecutados durante el quantum (TS_k) del procesador. Siguiendo el algoritmo, cada vez que se ha de asignar un nuevo trabajo (Ji) al sistema, se busca una fila con la misma cantidad de celdas libres que procesos tiene el trabajo a ser asignado. Una vez asignado el trabajo, se utiliza una política de *round-robin* para planificar las diferentes filas de la matriz. En este ejemplo, para un $n = 6$ un $MPL = 4$ y las condiciones mostradas,

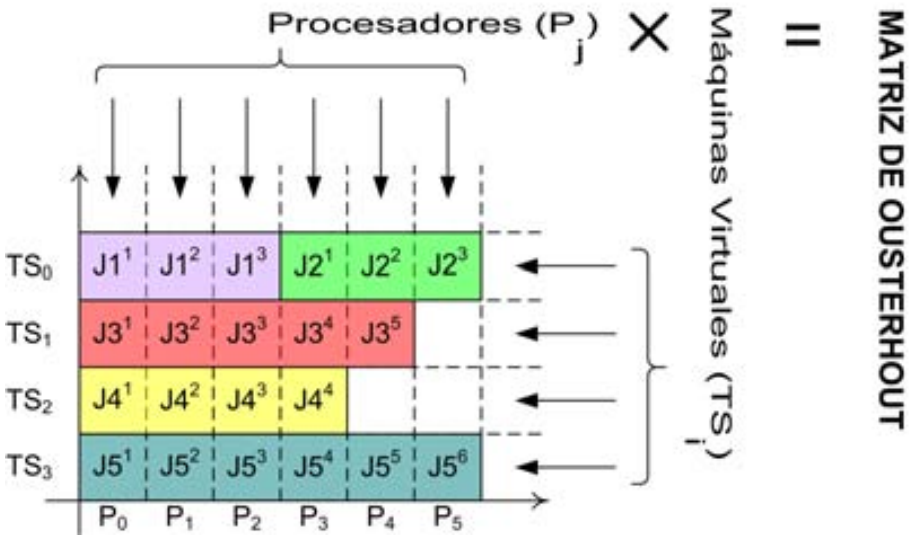

Figura 1.3: Matriz de Ousterhout

durante el quantum de tiempo TS_0 se ejecutarán los procesos pertenecientes a los trabajos $J1$ y $J2$, de manera que al finalizar dicho quantum se producirá un cambio de contexto global, de modo que el trabajo $J3$ será planificado en los procesadores P_0, P_1, P_2, P_3yP_4 durante el siguiente quantum TS_1, y así sucesivamente.

El término *Gang Scheduling* también ha sido empleado ampliamente en la literatura para referirse a la necesidad de la coplanificación. En [39] se define este término como un esquema de planificación que organiza sus tareas en grupos, de tal forma que las aplicaciones paralelas los conformen y que los grupos sean planificados simultáneamente en los procesadores de nodos diferentes. Esta definición concuerda completamente con lo ya establecido anteriormente por Ousterhout y utilizaremos este término para referirnos a ambas técnicas.

Una relajación del concepto del Gang Scheduling fue propuesta en [94], en el cual se establece que sólo es necesario coplanificar los procesos que están cooperando en un instante determinado.

Una posible forma de clasificar el extenso trabajo llevado a cabo en esta área es de acuerdo al método de control empleado para lograr la coplanificación de los procesos cooperantes, dicha clasificación quedaría conformada por los siguientes elementos:

- Coplanificación con control **explícito**: esta implementación de la coplanificación requiere de un cambio de contexto global simultáneo a lo largo de toda la máquina paralela [41]. Este enfoque es más apropiado

para ambientes dedicados y se ajusta a la definición de Gang scheduling.

- Coplanificación con control **implícito**: Las decisiones de planificación son tomadas por los planificador locales de acuerdo con la aparición de eventos locales o remotos. Los eventos pueden ser de comunicación, de memoria, de CPU, de actividad de usuarios locales o grado de multiprogramación (MPL). Como alternativas podemos citar la *coplanificación predictiva* [97, 94] basada en aumentar la probabilidad de coplanificación cambiando las prioridades de los trabajos en función de los eventos de comunicación recibidos y la *coplanificación dinámica* [13, 96], que planifica un proceso si recibe un evento de comunicación, expropiando la CPU al proceso en ejecución.

- Coplanificación con control **híbrido**: como su nombre indica, se hace uso de una combinación de las dos técnicas antes expuestas. Algunos de los resultados son: Buffered Coscheduling (BC) [84], Flexible Coscheduling (FCS) [40]y CoScheduling Cooperativo (CSC) [45].

Suele considerarse que las técnicas basadas en control híbrido son las que aportan más flexibilidad y facilidades de implementación. Lamentablemente la implementación de técnicas de coplanificación implica llamadas al sistema en modo de superusuario, para ser capaces de monitorizar los eventos de comunicación u otros. En este trabajo proponemos un nuevo acercamiento para solucionar este problema, que además considera las novedades tanto en los tipos de aplicaciones paralelas como en el hardware.

1.3. Tiempo Real estricto y débil

Los nuevos tipos de aplicaciones con características de tiempo real débil (*soft real-time*, ***SRT***) existentes en los clusters no dedicados motivan esta sección, en la cual introduciremos algunos modelos SRT y de tiempo real estricto (*real-time*, ***RT***). Es válido destacar que los sistemas SRT usualmente son considerados como una derivación o relajación de RT, como efectivamente ocurre. Debido a que nos centraremos en los sistemas SRT, en este trabajo colocaremos ambas teorías al mismo nivel en nuestro texto. La intención es profundizar en sistemas RT sólo lo necesario, dado el volumen de información existente y la orientación de nuestro trabajo.

1.3.1. Sistemas de Tiempo Real Estricto

Un sistema con requerimientos de tiempo explícitos, ya sea de naturaleza probabilística o determinista, es considerado de RT. La noción de prioridad

TIEMPO REAL		
Asignación de Prioridades	Fija	Dinámica
Algoritmo Representativo	RMS	EDF
Admisión de Peticiones	$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq n\left(2^{\frac{1}{n}} - 1\right)$	$\sum_{i=1}^{n} U_i \leq 1$
Tareas aperiódicas	Polling Server, Slack Stealing Algorithm	Total Bandwidth Server

Tabla 1.1: Resumen de los temas a tratar en la sección sobre sistemas tiempo real estricto.

es comúnmente utilizada para establecer orden en el acceso a recursos, tanto en la CPU como en la Red. La planificación de tareas RT será dividida en dos grupos de acuerdo a la forma en que tratan la prioridad, ya sea con prioridad **fija** y o con prioridad **dinámica**. La Tabla 1.1 muestra un resumen de las características y algoritmos que trataremos en esta subsección.

1.3.1.1. Planificación con Prioridad Fija

En el modelo de Planificación con Prioridad Fija todas las tareas de un mismo trabajo tienen la misma prioridad, que no cambia en el tiempo. La nomenclatura usualmente empleada denomina a cada tarea como τ_i, dónde i es la prioridad de la tarea. Una tarea es *periódica* (Figura 1.4) si ocurre cada cierto intervalo regular de tiempo, siendo la longitud entre los sucesivos arribos de los trabajos que componen la tarea τ_i constante, llamado el *período* de la tarea y denominado T_i. Cabe destacar que la prioridad i se calcula como la inversa del período ($i = \frac{1}{T_i}$). El *deadline* (plazo) de una tarea periódica se define como D_i, representando este valor el máximo valor de tiempo que puede transcurrir antes de que el trabajo i de la tarea τ_i consuma su *tiempo de cómputo* (C_i).

El modelo inicialmente propuesto en la teoría RT (conocido como Modelo de Liu y Layland [72]) asume que:

1. Todas las tareas son periódicas
2. Todas las tareas llegan al inicio de su período y tienen un deadline igual a su período.
3. Todas las tareas son independientes, es decir, no tienen relaciones de precedencia en relación a los recursos que utilizan.

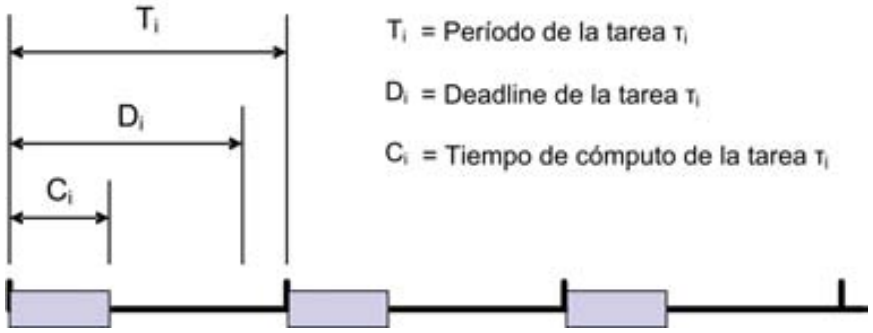

Figura 1.4: Esquema de una Tarea Periódica.

4. Todas las tareas tienen un tiempo de cómputo fijo, o al menos limitado, que es menor o igual que su período.
5. Ninguna tareas se puede suspender a si misma.
6. Todas las tareas son completamente desalojables.
7. No se consideran *overheads*, relacionados con el sistema operativo.
8. Solo existe un procesador.

Bajo este modelo, las tareas de los trabajos periódicos ocurren a lo largo del tiempo a intervalos regulares de longitud constante T_i (el período de la tarea). Cada tarea tiene un deadline D_i unidades de tiempo después de su liberación. Llamamos *rígida* a una tarea de tiempo real (*hard real-time*) si debe cumplir su plazo (tanto a nivel de tiempo de comienzo como de final); de no cumplirlos se producirán daños no deseados o un error fatal en el sistema.

Una tarea RT es llamada *flexible* si tiene un plazo asociado, que es conveniente, pero no obligatorio; aunque haya vencido el plazo, aún tiene sentido planificar y completar la tarea.

Una tarea *aperiódica* debe comenzar o terminar en un plazo o bien, puede tener tanto una restricción para el comienzo como para la finalización.

Los *análisis de admisión* son empleados para predecir si las restricciones temporales de una tarea serán satisfechos en tiempo de ejecución. Los que tienen en cuenta todos los elementos necesarios (test suficiente y necesario) alcanzan complejidad NP completa, por lo que son impracticables. Son generalmente de menor complejidad algorítmica los tests que son suficientes pero no necesarios. Los tests suficientes pero no necesarios tienen la desventaja de que son pesimistas.

El hecho de que las prioridades no varíen hace más efectivos los análisis de admisión, siendo el Teorema del Instante Crítico [72] el empleado en este

caso. El instante crítico para una tarea es el tiempo de liberación para el cual el tiempo de respuesta es el máximo (o excede su deadline, para el caso en el cual el sistema está tan sobrecargado que los tiempos de respuesta crecen sin límites). Este teorema establece que, para un conjunto de tareas periódicas con prioridades fijas, el instante crítico de una tarea ocurre cuando es invocada simultáneamente con todas las tareas de mayor prioridad que ella. El intervalo de 0 a D_i es entonces uno en el cual la demanda de tareas de mayor prioridad $\tau_1 ... \tau_{i-1}$ está en un máximo, creando la situación más difícil para que τ_i cumpla su deadline. Este teorema ha probado ser robusto, siendo verdadero incluso cuando muchas de las restricciones antes listadas son relajadas.

El grupo de políticas de asignación de trabajos con prioridad fija es conocido como RMS (*Rate-Monotonic Scheduling*), en el cual a la tarea con el menor período se le asigna la mayor prioridad, a la próxima tarea de menor período la siguiente prioridad y así sucesivamente. Se ha probado que para un conjunto de n tareas periódicas con política de asignación RMS la asignación es posible si:

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq n \left(2^{\frac{1}{n}} - 1\right) \tag{1.1}$$

Como ejemplo podemos decir que un par de tareas es viable si su utilización de CPU combinada no excede el 82,84 %. Si n tiende al infinito, el valor $n \left(2^{\frac{1}{n}} - 1\right)$ se aproxima a $\ln(2)$ para un valor aproximado de utilización de 69,31 %. Muchas veces se asume que bajo estas condiciones el valor antes mencionado defina la máxima utilización posible, lo cual es errado, ya que esto es solo una condición de suficiencia. Este límite es cerrado en el sentido de que existe algún conjunto de tareas inviables cuya utilización arbitrariamente se acerca a $n \left(2^{\frac{1}{n}} - 1\right)$. Por este motivo es posible encontrar multitud de conjuntos de tareas con utilización mayor que el 69,31 %. En [65] encontramos un interesante estudio sobre RMS y los niveles de utilización de esta técnica, que como promedio es del 88 %.

No obstante a su utilidad, los análisis de admisión tienen limitaciones, como lo son:

1. La condición de admisión es necesaria pero no suficiente (es decir, pesimista).
2. Impone restricciones poco reales a las características de la tareas, es decir $D_i = T_i$.
3. Las prioridades de la tareas han de ser asignadas utilizando RMS, caso contrario el análisis de admisión es insuficiente.

Por estas razones se han desarrollado pruebas de admisión más complejas, pero que no tienen las limitaciones antes expuestas. En [20] se propone una prueba de admisión de complejidad polinomial menos pesimista que el representado por la Fórmula 1.1. Esta prueba (Ecuación 1.2) ha demostrado ser fuerte.

$$\prod_{i=1}^{n} \left(\frac{C_i}{T_i} + 1 \right) \leq 2 \tag{1.2}$$

Existen también tareas, denominadas ***aperiódicas*** que no cumplen con los requerimientos del modelo anterior y han de ser contempladas también en los modelos de RT. Este tipo de tareas puede ser diferente de las periódicas en que los tiempos de arribo o de cómputo sean significativamente diferentes, que no tengan deadlines estrictos o bien alguna combinación de las características antes expuestas.

Si el modelo de tareas periódicas es ligeramente relajado, siendo C_i el máximo tiempo de ejecución y T_i el tiempo mínimo entre los arribos, el modelo [72] sigue siendo válido. Sin embargo es poco práctico y eficiente hacer reservas si los tiempos de cómputo o de arribo de las tareas aperiódicas son muy variables.

Una solución posible al problema de las tareas aperiódicas es implementar un servidor que se ejecute como una tarea periódica normal y que se encargue de ejecutar las tareas aperiódicas. Esta técnica en la literatura se denomina *Polling Server* [89]. La capacidad del servidor se calcula off-line y en la mayoría de los casos se asigna el mayor tiempo de cómputo, que permita el análisis de admisión. En tiempo de ejecución, el servidor se ejecuta periódicamente y su tiempo de cómputo se emplea en ejecutar las tareas aperiódicas. Una vez consumido su tiempo de cómputo su ejecución se suspende hasta su próximo arribo programado, también periódico. Como el servidor se comporta como una tarea periódica, los análisis de admisión diseñados para ellas se pueden aplicar normalmente.

Para las tareas aperiódicas los polling servers significan una mejora sustancial respecto al procesamiento en background. Lógicamente, si llegan demasiadas tareas aperiódicas la capacidad del servidor se verá sobrecargada y algunas tareas tendrán tiempos de respuesta peores. El caso inverso también ocurre, si no llegan tareas aperiódicas la capacidad de cómputo reservada al servidor se infrautiliza. Una posible solución a este último problema es variar la prioridad del servidor de acuerdo a si tiene o no tareas pendientes [104].

Partiendo de la idea anterior, servidores que se ejecutan como tareas periódicas, se han realizado varios trabajos germinales para mejorar el procesamiento de tareas aperiódicas en entornos RT. Entre los más interesantes están el algoritmo de *Slack Stealing* [106], que es óptimo en el sentido de que

minimiza el tiempo de respuesta para las tareas aperiódicas manteniendo los deadlines de todas las tareas RT.

Finalmente mostraremos los resultados encontrados en la literatura para **RT** en **multiprocesadores**. Vemos reflejadas dos aproximaciones a la planificación de tareas RT en múltiples procesadores, particionada y global. En la aproximación por particiones cada tarea es asignada estáticamente a un procesador y en la global las tareas compiten por el uso de los procesadores. En [30] se muestra que la planificación global de tareas para m-procesadores utilizando RMS de un sistema de $m+1$ tareas no puede ser garantizado para utilizaciones del sistema por encima de 1. Por otro lado, utilizando particionado con RMS *next-fit* podemos garantizar la viabilidad de los sistemas de tareas para utilizaciones por encima de $m/(1+2^{1/3})$. Este límite es conocido como el efecto Dhall, en referencia al investigador que lo determinó.

Dado que el problema de particionamiento óptimo de tareas entre múltiple procesadores es de tipo NP completo, las soluciones óptimas es posible solo para los casos más simples. Por lo tanto, han de usarse heurísticas para encontrar soluciones aproximadas, siendo la más empleada RMFF (*Rate Monotonic First-Fit*). Para este caso, se ha determinado que la máxima utilización del sistema es de $(m+1)\left(2^{1/(m+1)}-1\right)$. Otro resultado interesante es que para una planificación con prioridades fijas en un sistema multiprocesador de m nodos, sin importar si es global o particionado o el esquema de asignación de prioridades, la utilización garantizada no puede ser mayor que $(m+1)/2$ [10].

1.3.1.2. Planificación con Prioridad Dinámica

Planificando con prioridades estáticas, todas las tareas pertenecientes a un mismo trabajo tienen la misma prioridad, si empleamos prioridades dinámicas no será así. Otorgando las prioridades de forma dinámica, cada trabajo perteneciente a una tarea tiene diferentes prioridades, en función de cuan cerca esté su deadline. Uno de los algoritmos con prioridad dinámica más estudiados es el EDF (*Earliest Deadline First*).

EDF es un algoritmo dinámico que no se requiere que el tiempo de ejecución por ráfaga de CPU de los procesos sea igual, como si ocurre con RMS. Cada vez que un proceso necesita la CPU, anuncia su presencia y su plazo. El planificador mantiene una lista de los procesos ejecutables en orden por plazo. El algoritmo ejecuta el primer proceso de la lista, el que tiene el plazo más cercano. Cada vez que un nuevo proceso está listo, el sistema verifica si su plazo se va a cumplir antes que se cumpla el del proceso que se está ejecutando. En tal caso, el nuevo proceso expropiará al actual. Para este algoritmo, suponiendo las condiciones del Modelo de Liu y Layland, la prueba

de admisión para un conjunto de n tareas periódicas se establece por la utilización del procesador 1.3.

$$\sum_{i=1}^{n} U_i \leq 1 \tag{1.3}$$

En esta ecuación, el *nivel de utilización*, denotado como U_i se define como $U_i = \frac{C_i}{T_i}$.

Aunque existe otro algoritmo que emplea prioridad dinámica, denominado LLF (*Least Laxity First*)[79], este introduce un mayor overhead al sistema, razón por la cual la mayor parte de la investigación se centra en mejorar el algoritmo EDF. Las mejoras se centran en mejorar los análisis de admisión y en relajar algunos de los postulados simplistas del algoritmo. En [27] se muestra que el algoritmo EDF es óptimo en el sentido de que si existe un algoritmo que puede construir una planificación viable en un solo procesador, entonces el algoritmo EDF también puede construir una planificación viable.

Cuando se utiliza EDF el análisis de admisión puede ser realizado teniendo en cuenta el criterio de *Demanda de Procesador*. La demanda se calcula para un conjunto de trabajos RT y un intervalo de tiempo $[t_1, t_2)$ como

$$h_{[t_1,t_2)} = \sum_{t_1 \leq r_k, d_k \leq t_2} C_k. \tag{1.4}$$

Es decir, la demanda de procesador es el valor representado por la cantidad de tiempo de cómputo pedido por todos los trabajos con tiempo de arribo en o después de t_1 y deadline antes o en t_2. A partir de este valor, el análisis de admisión puede ser efectuado considerando que la demanda de procesador no puede superar el tiempo disponible, es decir, podemos establecer la viabilidad teniendo en cuenta 1.5.

$$\forall t_1, t_2 \quad h(t_1, t_2) \leq (t_2 - t_1) \tag{1.5}$$

Como las prioridades son dinámicas, la planificación de tareas aperiódicas mejora ya que puede reaccionar mejor al arribo de una tarea no periódica. Uno de los principales enfoques es el del *Total Bandwidth Server* (TBS) [99], que es una de las técnicas más eficientes para planificar tareas aperiódicas bajo EDF. TBS funciona asignando a cada trabajo aperiódico un deadline de tal forma que la carga total aperiódica no exceda un valor máximo U_s. El deadline asignado se calcula mediante la expresión asociada 1.6, nótese que esta toma en cuenta las asignaciones a tareas anteriores (representadas por d_{k-1}). Una vez asignado el deadline, el requerimiento es insertado en el sistema como el de cualquier otra tarea periódica, pero respetando el

umbral U_s antes establecido. Podemos afirmar que dado un conjunto de n tareas periódicas con una utilización del procesador de U_p y un TBS con utilización U_s todo el conjunto es viable para su planificación si y solo si $U_p + U_s \leq 1$. Es válido mencionar que el proceso de asignación de deadlines puede ser optimizado para minimizar el tiempo de respuesta a las aplicaciones aperiódicas [22]. En esta aproximación, el *ancho de banda* del servidor define como un umbral (U_s), que representa la capacidad de cómputo disponible respetando las tareas periódicas.

$$d_k = max\,(r_k, d_{k-1}) + \frac{C_k}{U_s} \tag{1.6}$$

La Figura 1.5 muestra un ejemplo de uso de un TBS. Dos tareas periódicas con períodos $T_1 = 6$, $T_2 = 8$ y tiempos de cómputo $C_1 = 3$, $C_2 = 2$ respectivamente se planifican bajo EDF para una utilización $U_p = 0,75$ implicando que el ancho de banda del servidor disponible es de 0,25 (calculado mediante $U_s = 1 - U_p$). Los deadlines son calculados utilizando la Ecuación 1.6 en el momento de los arribos de las tareas aperiódicas. El primer arribo de tarea aperiódica ocurre en $t = 3$ y se le asigna un deadline $d_1 = 7$, como $d_1$es el deadline más cercano a expirar globalmente es servido inmediatamente. La próxima tarea aperiódica arriba en $t = 9$ y recibe un deadline $d_2 = 17$, pero no es servida de forma inmediata, ya que en ese momento está en ejecución una tarea con deadline más urgente (τ_2, con deadline en $t = 16$). Por último llega una tarea aperiódica en $t = 14$ que recibe un deadline $d_3 = 21$, que no es servida de forma inmediata ya que en el momento de su llegada la tarea periódica τ_1 está activa y tiene un deadline más bajo.

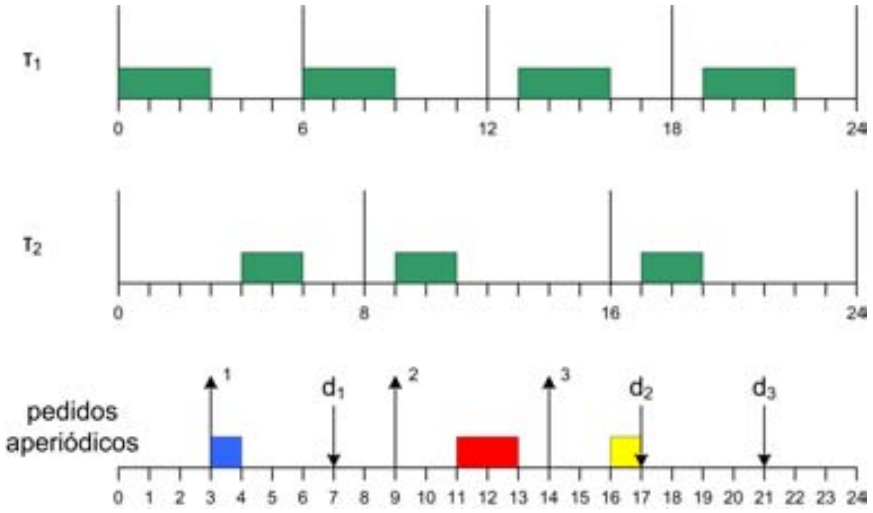

Figura 1.5: Ejemplo de planificación bajo EDF empleando *Total Bandwidth Server*.

Para la **implementación de reservas** bajo EDF disponemos como una opción válida del algoritmo *Constant Bandwidth Server* (CBS) [5]. Un CBS se caracteriza por un presupuesto c_s, un deadline dinámico d_s y un par ordenado (Q_s, T_s), donde Q_s es el presupuesto máximo y T_s el período del servidor. Llamamos a el cociente $U_s = Q_s/T_s$ el ancho de banda del servidor. A cada trabajo servido por el CBS se le asigna un deadline conveniente e igual al deadline actual del servidor, calculado para no sobrepasar el ancho de banda reservado. Mientras el trabajo se ejecuta, el presupuesto c_s es decrementado en el tiempo consumido por el trabajo. Cada vez que $c_s = 0$ se recarga el presupuesto del servidor a Q_s y el deadline del servidor se pospone en T_s, para reducir la interferencia a otras tareas.

1.3.2. Sistemas Tiempo Real Débil

La teoría para RT está concebida tomando como axioma que la pérdida de un deadline ha de considerarse un fallo en el sistema. Para lograr respetar de forma estricta los deadlines de la tareas, la teoría RT se basa en la formulación estricta del peor caso. Este enfoque permite tener una cota superior para la carga en cualquier instante de tiempo, lo que nos permite conocer si los deadlines de las tareas se cumplirán.

Pero qué ocurre si el peor caso no está cerca del caso promedio, como sucede en la mayoría de los sistemas de control para lo que está diseñada la teoría RT estricta. Un ejemplo claro y ampliamente utilizado en la literatura para ejemplificar esta situación es el de un vídeo online. Si no hay cambios bruscos en las escenas, los *frames* transmitidos son de menor tamaño, debido a la codificación de la información en frames I, P o B. Esto tiene como consecuencia que la media de uso del ancho de banda de red sea bastante menor, diferencia que puede llegar a ser de varios órdenes de magnitud. Si tratamos este caso de forma estricta, deberíamos de reservar recursos que una parte importante del tiempo estarían ociosos.

Un ejemplo sencillo lo encontramos en la caracterización del uso de ancho de banda de red durante una vídeo conferencia, donde podemos encontrar que los valores oscilen entre una media de 500-600 kbps cuando los participantes de la vídeo conferencia están quietos, y hasta 3500-4000 kbps si se mueven frente a la cámara de vídeo. La variaciones se deben al algoritmo utilizado, basado en enviar sólo las diferencias con la escena anterior, lo que motiva que la cantidad de información a transmitir en cada caso se diferente. Resulta evidente que para esta situación reservar el peor caso es un desperdicio importante de recursos, esta sección intenta profundizar en la teoría relacionada con el tratamiento de estos casos, la teoría de SRT.

1.3.2.1. Aplicaciones SRT

El caso anteriormente descrito presenta una situación que cada vez es más común, tanto para usuarios locales como para los usuarios paralelos. En esta sección mostraremos varios ejemplos de aplicaciones, tanto locales como paralelas, que requieren especial atención dados sus requerimientos de recursos periódicos.

En [35] se estudian varios métodos para identificar aplicaciones de tipo *Human Centered* (HuC), i.e.: reciben el foco de atención del usuario local, razón por la cual, según se plantea en ese trabajo, deberían de recibir especial atención. La caracterización de las aplicaciones estudiadas en este trabajo nos permite conocer mejor las diferencias en los requerimientos de recursos de las aplicaciones a lo largo de un lapso de tiempo significativo para nuestros objetivos. El cambio más significativo se refleja en el paso de aplicaciones con interactividad, basada en tiempos de respuesta del teclado o el ratón, como a los editores de texto de diferentes tipos y a las aplicaciones multimedia. Estas últimas necesitan la CPU de forma periódica para su correcta ejecución, y podría ocurrir que durante largos períodos de tiempo no reciban ningún evento originado por el usuario, como un clic de ratón u otros. Aplicaciones de estas características (mayores requerimientos periódicos de recursos) componen el grupo de aplicaciones que denominaremos aplicaciones locales SRT (*local_ SRT*).

Otros componentes de este grupo son los tipos de juegos con algoritmos de visualización complejos, como los conocidos por *First Person Shooter* (FPS). Es una tendencia que los juegos de ordenador consuman cada vez más recursos y en caso de no recibirlos de forma periódica, su ejecución no sea satisfactoria para el usuario. Este tipo de aplicación se emplea en los estudios como aplicación comparativa.

Por otro lado encontramos que es cada vez más común que los usuarios paralelos necesiten ejecutar aplicaciones con necesidades temporales. Este es el caso descrito en [110], donde rutinas de detección de obstáculos en secuencias de *frames* hacen que el volumen de cálculo sea alto y de acuerdo a la finalidad del resultado, ha de obtenerse con urgencia. Este tipo de aplicaciones requiere de hardware especializado o de cómputo paralelo de altas prestaciones. También en [85] encontramos un caso novedoso de aplicación paralela. En este estudio encontramos un tipo de aplicaciones con una alta cantidad de eventos, generados por instrumentación científica, y una ausencia casi total de usuarios. Para lograr recolectar todos los eventos necesitamos que las tareas que se generan con cada evento gocen de prioridades en el sistema.

Otro ejemplo de aplicación paralela SRT, es la representada por una aplicación paralela con tiempo de turnaround acotado, que representa el caso en el que usuario paralelo necesita los resultados de la ejecución de su aplicación

paralela dentro de un intervalo de tiempo específico. Las aplicaciones que requieran de tiempos de cómputo periódico en diferentes nodos de un sistema distribuido, ya sea dedicado o no, recibirán en este estudio el tratamiento de aplicaciones paralelas SRT (*par_SRT)*.

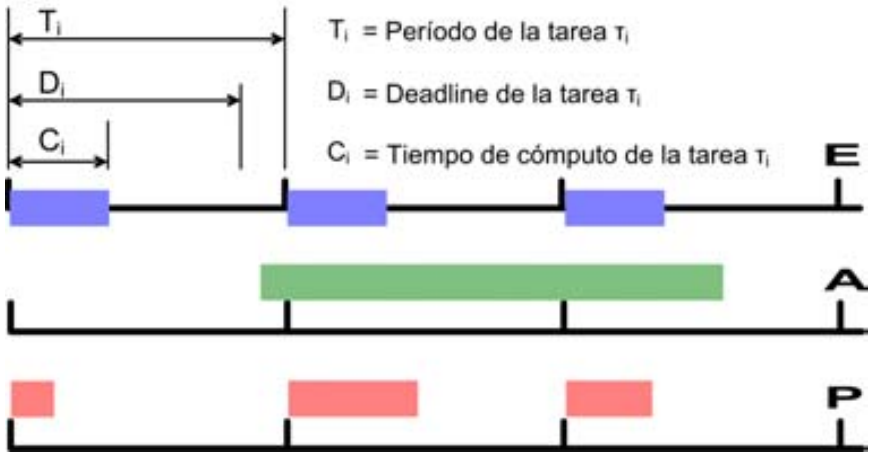

Figura 1.6: Tiempo de cómputo de aplicaciones periódicas y aperiódicas.

La Figura 1.6 contiene el patrón de uso de la CPU de aplicaciones periódicas (caso P) y aperiódicas (caso A), y el modelo clásico de tareas RT (caso E) con fines comparativos. Las aplicaciones paralelas con tiempo de turnaround acotado son aplicaciones aperiódicas, que arriban al sistema en un tiempo cualquiera y pueden emplear la CPU siguiendo patrones no periódicos.

Por otra parte, al decodificar un vídeo comprimido en formato mpeg encontramos frames que puedan requerir diferentes tiempos de CPU para su decodificación, aunque de forma periódica. Esta situación está representada en la Figura 1.6 por el caso P, y representa un caso modelo a tratar en este trabajo, relacionado con las tareas locales SRT. La carga local de tipo SRT estará representada por el reproductor multimedia Xine [2], que presenta patrones de uso similares al descrito anteriormente.

1.3.2.2. Modelos SRT

¿Cómo representamos estas tareas de tipo SRT, ya sea locales o paralelas, en forma de modelos? Han existido muchos enfoques basados en la teoría RT existente, como lo es mezclar tareas periódicas con aperiódicas sin deadlines o complejos modelos que le asignan a cada tarea un *valor de utilidad* en función de la QoS requerida por la tarea. El disponer de modelos para este caso nos permite predecir, calcular e incluso garantizar algún recurso a este tipo de tareas. Describiremos algunos de los modelos (Figura 1.7) encontrados en la literatura.

Figura 1.7: Taxonomía: Modelos SRT.

El modelo basado en demoras (*lateness*) se formula asociando a los deadlines de las tareas SRT una restricción que representa la demora permisible. Esta restricción puede tener varias formas e incluso podemos encontrarla en forma compuesta. Por ejemplo, si definimos $\alpha(x)$ como la parte de los trabajos que pierden su deadline por más de x unidades de tiempo, entonces se suele definir la demora en la forma $\alpha(x) \leq \beta$. Esta notación representa una restricción en el sentido de que limitamos la cantidad de trabajos que pierden su deadline a β. En este modelo, cada deadline perdido se considera un fallo y β limita la fracción de trabajos que pueden fallar. El valor $\alpha(-\infty)$ representa la cantidad de fallos, incluidos los rechazados en los análisis de admisión. En general, para un conjunto de valores de tiempo denotado por $\{x_1, \ldots, x_m\}$ y una lista de restricciones $\{\beta_1, \ldots, \beta_m\}$ podemos requerir que $\alpha(x_i) \leq \beta_i, 1 \leq i \leq m$. Esta especificación de las restricciones nos permite tener en cuenta la naturaleza estocástica de los tiempos de arribo y de cómputo de las tareas tipo SRT, lo cual a su ves nos lleva a la formulación del concepto de *viabilidad del caso promedio*, es decir, el mayor monto de carga promedio que el sistema puede procesar cumpliendo las restricciones de demora.

También basados en la naturaleza estocástica de las aplicaciones SRT se han formulado otros modelos, como el Modelo Estocástico RMS, *Stochastic RMS* (SRMS) [14, 15]. Este modelo está especialmente designado para ser utilizado en sistemas en los cuales las tareas periódicas tienen tiempos de cómputo y requerimientos de QoS altamente variables, además de que los deadlines de las tareas sean débiles, no duros. Este último requerimiento significa que algunos deadlines pueden perderse, aunque con restricciones en las pérdidas. El diseño del algoritmo SRMS también se pensó de tal forma que maximice el uso de los recursos a la vez que minimice el uso de recursos de las tareas que pierden sus deadlines.

Por último mencionaremos los modelos que consideran que todos los parámetros de las tareas SRT son estocásticos, es decir, que tienen tiempos de arribo entre las tareas, tiempos de cómputo y deadlines estocásticos. Aunque en este caso es difícil derivar un análisis completo de la viabilidad para el caso

promedio, se han desarrollado modelos para calcular la fracción de tareas demoradas (*late tasks*) para los casos con tráfico pesado, en los cuales los modelos tienen altos niveles promedio de utilización. Este método es conocido como RTQT (*Real-Time Queueing Theory*) [33, 66, 67], Teoría de Colas para Tiempo Real debido a que es una extensión de la Teoría de Colas que tiene en cuenta de forma explícita los requerimientos temporales de las tareas SRT. RTQT asume que las tareas se planifican bajo EDF y su métrica de rendimiento se calcula en base a la fracción de las tareas que terminan dentro de su deadline.

Este algoritmo ha de mantener información del tiempo restante hasta que el deadline de cada tarea finalice (*lead time*). Este requerimiento combinado con la necesidad del algoritmo EDF de mantener información de los deadlines de cada tarea, hace que este problema sea analíticamente intratable. El problema se resuelve parcialmente ya que se pueden obtener buenas aproximaciones para el caso de tráfico pesado, que servirá de cota para cualquier otro caso más ligero. En RTQT este caso se alcanza cuando ($\rho = 1$), donde ρ es la intensidad del tráfico e intenta significar el momento de mayor necesidad de cómputo a través del momento en el que llegan más tareas.

1.3.3. Los sistemas operativos comerciales y el Tiempo Real

Una de las vertientes que más esfuerzos ha concentrado es intentar adaptar los sistemas operativos de uso común al tiempo real. Esta idea, aunque atractiva en algunos sentidos, tiene el inconveniente de que los Sistemas Operativos de Propósito General (*General Purpose Operating Systems*, ***GPOS***) están diseñados de acuerdo a los siguientes objetivos:

1. *Equitatividad*: todos los trabajos han de compartir los recursos del GPOS de forma equitativa.
2. *Eficiencia*: los recursos del GPOS deben ser empleados para maximizar el rendimiento y para lograr el mejor tiempo medio de respuesta para *todos* los trabajos.
3. *Facilidad de Uso*: los GPOS han de ser fáciles de usar, y en lo posible ser capaces de evitar errores accidentales de parte del usuario.

Al enfrentarnos al problema de la planificación de aplicaciones tiempo real, estos objetivos más que ayudarnos entorpecen nuestro trabajo, debido a que la equitatividad y la eficiencia no son tan importantes como lograr cumplir las restricciones de tiempo de las tareas tiempo real.

Entre los trabajos que desarrollan esta idea, podemos destacar [69], que permite la ejecución de aplicaciones tiempo real en el espacio de usuario en

Windows NT y su correspondiente versión para Linux [26]. La arquitectura en términos generales está desarrollada a través de un *dispatcher* a un nivel más protegido y un agente accesible a los usuarios. Este trabajo se basa en la extensión POSIX.4 y sus reglas de planificación por prioridades. En [70] podemos encontrar un desarrollo más actual de esta arquitectura a dos niveles, pero con algo más de potencial, pues el agente, llamado *Allocator* para este trabajo, permite la implementación de diferentes políticas de planificación de tareas RT y SRT.

Por otro lado, centrándose en las aplicaciones de tiempo real débil, encontramos una versión "mejorada" de Linux, denominada Linux-SRT [25]. En este trabajo se provee a Linux de un sistema para la planificación predecible y la administración de la QoS. Los recursos que controla son la CPU y el ancho de banda de los discos duros. También en [86, 55] se intenta garantizar una QoS determinada a las tareas SRT y RT e incluso conversión entre las reservas de los diferentes tipos de tareas.

En [46] encontramos una aproximación más completa, en este caso enfocada en los sistemas empotrados. En este trabajo, no solo se controlan los recursos usuales (CPU y memoria principal), sino que además se propone una arquitectura paralela para el control de la Entrada/Salida y de esta forma poder garantizar este recurso a las tareas.

Aunque el enfoque de dotar a los GPOS de capacidades para la planificación de aplicaciones RT y SRT ha sido ampliamente desarrollado, en nuestro caso es poco práctico. Los administradores de redes son poco propensos a permitir la modificación de los kernels de Linux en sus sistemas o a instalar versiones de Linux no estables, por lo que esta opción carece de interés para nosotros. Cabe destacar una de las conclusiones más importantes que se desprenden del estudio de estos trabajos. Linux, el OS empleado durante este trabajo, fue designado como un GPOS, por lo que tiene las siguientes deficiencias:

- Baja apropiatividad: pese a los esfuerzos invertidos para mejorarla, el kernel de Linux aún no puede garantizar el desalojo de tareas.
- Brinda un pobre soporte para tareas RT, pues solo permite definirles prioridades muy altas, lo que trae como consecuencia que la capacidad de respuesta del sistema se ve seriamente afectada.
- Carece del concepto de período, deadline y tiempo de cómputo, elementos esenciales de la teoría RT. Tampoco se puede hacer reservas de recursos.
- La frecuencia del reloj no tiene la granularidad adecuada para tiempo real.

1.4. La era de los procesadores multi-core

Como hemos mencionado anteriormente, el crecimiento en el rendimiento de los procesadores se ha visto frenado por las barreras físicas del espacio y la velocidad de las señales. Los fabricantes de procesadores, en respuesta a esta situación, han optado por incluir más núcleos en el mismo procesador; siendo prácticamente imposible comprar un ordenador en la actualidad que no disponga de al menos dos núcleos de procesamiento. En los procesadores, cada uno de estos núcleos son unidades de ejecución completas, combinándose con la cache y sus controladores.

Cabe destacar que en la actualidad los procesadores multi-core son un producto establecido en el mercado actual, con las principales compañías produciéndolos [29, 57]. Por esta razón, nos vemos obligados a expandir nuestro entorno para la planificación de aplicaciones paralelas para aprovechar las nuevas potencialidades de los nodos de los clusters no dedicados. En esta sección mostraremos algunas de las razones que nos impulsan a centrarnos en la planificación temporal, como lo son los problemas introducidos en los GPOS debido al mal uso de la cache o la optimización del consumo de energía.

1.4.1. Cache

La cache en un procesador es un aspecto de especial interés debido a los compromisos que conlleva, ocasionados por su elevado coste económico y la necesidad de usarla eficientemente. En esta sección mostramos algunas de las arquitecturas de cache implementadas por las dos empresas líderes en la producción de procesadores, luego mostramos el estado de arte.

1.4.1.1. Arquitecturas

En esta subsección intentaremos mostrar algunas de las arquitecturas de cache más comunes para procesadores multi-core en la actualidad, centrándonos principalmente en los dos mayores productores de procesadores a nivel mundial, Intel y AMD. La selección de estos fabricantes, aunque excluye a otros con propuestas de procesadores muy interesantes, como IBM, tiene una razón muy sencilla. Aunque muchos de los MPP incluidos en la lista de los más rápidos a nivel mundial tienen procesadores IBM u otros, nuestro escenario es muy diferente, como hemos mencionado antes. Nos centramos en NOWs tipo COTS, donde lo más usual es encontrar que las estaciones de trabajo tienen procesadores Intel o AMD.

La Figura 1.8 muestra los dos casos más comunes y con más posibilidades de encontrarse en nuestro escenario. A la izquierda un procesador dual-core en el cual no se comparten caches a ningún nivel, y a la derecha otro en el

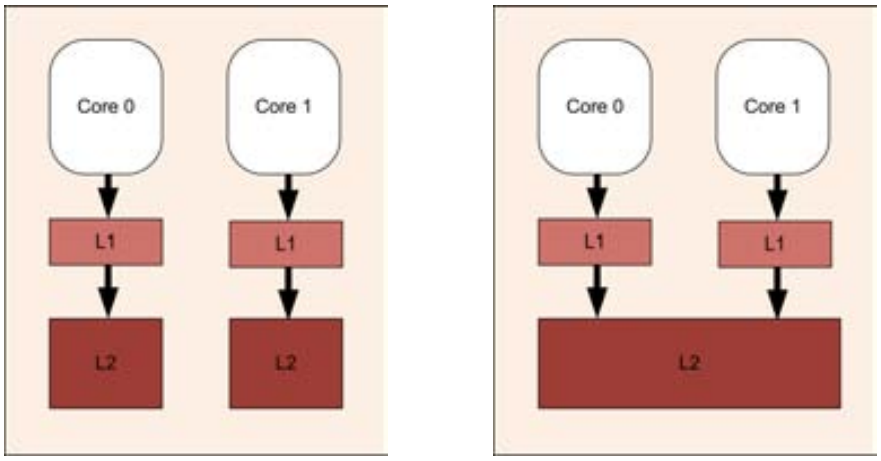

Figura 1.8: Arquitecturas de cache de procesadores multi-core, con y sin la L2 compartida.

que se comparte la L2. La Tabla 1.2 resume algunos ejemplos de los casos antes mencionados.

Nombre	Nombre en Código	Tamaño L2, total	L2 Compartida	SOI (nm)	Fab.
Athlon 64 X2	Manchester	1 MB	no	90	AMD
Athlon II	Regor	2 MB	no	45	AMD
Turion X2	Tyler	1 MB	no	65	AMD
Pentium D	Presler	4 MB	no	65	Intel
Core Duo	Yonah	2 MB	si	65	Intel
Core 2 Duo	Allendale	4 MB	si	65	Intel

Tabla 1.2: Ejemplos de procesadores dual-core y con y sin L2 compartida.

La Figura 1.9 muestra la arquitectura de cache empleada por Intel en los procesadores Itanium y en lo que actualmente es su gama más actual de procesadores, Intel Core i5 e i7. Cabe destacar que Intel también introduce el concepto de las SmartCache, cuyo uso es dinámico, en dependencia de las necesidades de los cores, pudiendo ocurrir que un core llene toda la cache compartida, en dependencia de las necesidades de cada core. En estos casos las L2 son relativamente pequeñas, de 256 KB por core; y las L3 muy grandes, llegando a alcanzar hasta los 12 MB.

En el caso de AMD, emplea esta arquitectura de cache en sus procesadores para servidores, los Opteron, y en su línea de procesadores quad-core, los Phenom.

Finalmente mostramos dos arquitecturas de cache, Figuras 1.10 y 1.11 empleadas en los procesadores para servidores de Intel de cuatro (Intel Xeon

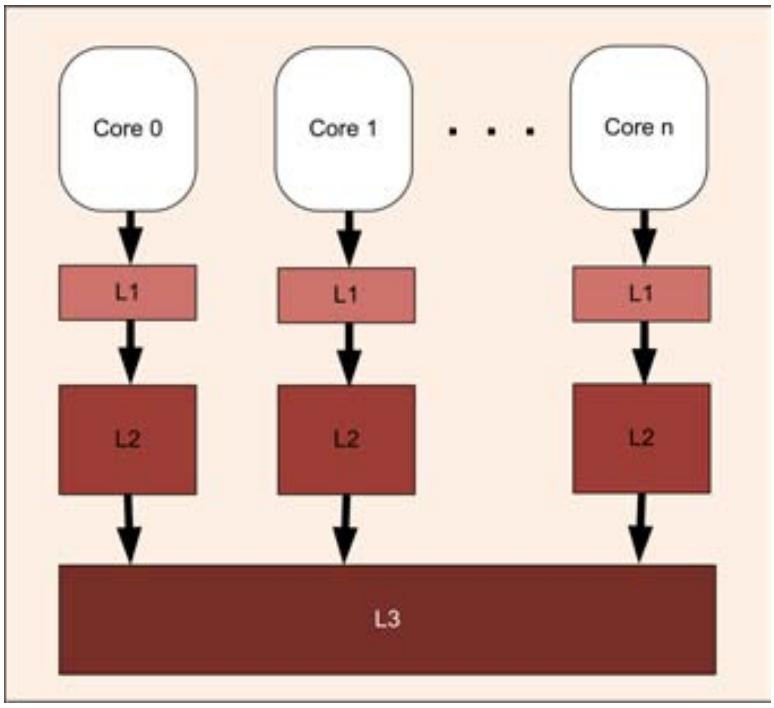

Figura 1.9: Arquitecturas de cache de procesadores multi-core, L3 compartida.

Quad-Core, Tigerton) y seis cores (Intel Xeon 74xx, Dunnington) respectivamente.

La arquitectura mostrada en la Figura 1.10 también ha sido empleada en el Intel Core 2 Quad.

1.4.1.2. Contención de recursos y co-ejecución dependiente

La gestión de los buses de memoria es analizada en [68], trabajo que contiene un estudio preliminar sobre los problemas relacionados con el uso del bus de memoria compartido por los procesadores que componen un procesador SMP. La conclusión principal es que solo podemos colocar varios procesos RT o SRT en un procesador SMP si controlamos el ancho de banda del bus de memoria disponible y su uso. Proponen que las aplicaciones RT o SRT modelen su comportamiento de acceso a memoria en ráfagas, para facilitar la planificación del uso del bus de memoria.

Por otra parte en [62] se muestra con resultados experimentales el impacto negativo de la contención en los buses de memoria compartidos en procesadores SMP. Para solucionar este problema proponen un planificador que tome en cuenta los problemas de contención en el bus de memoria.

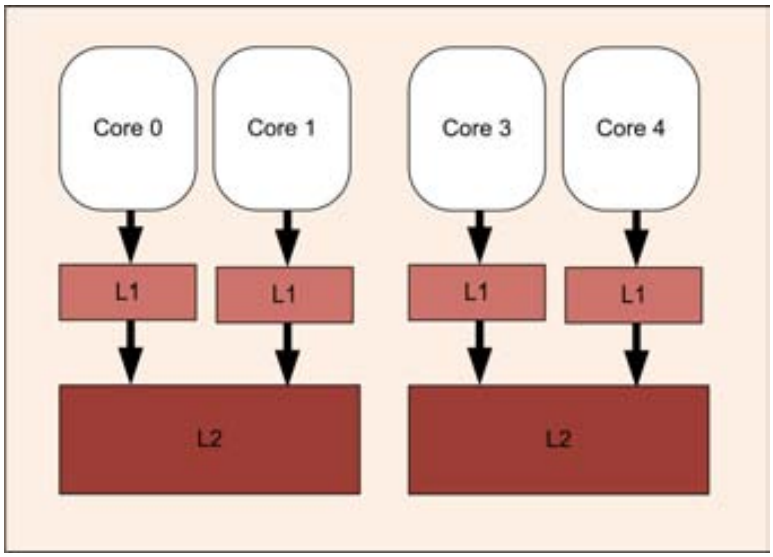

Figura 1.10: Arquitecturas de cache de procesadores multi-core, L2 parcialmente compartida.

Los planificadores diseñados para las CPU convencionales, dan por sentados ciertos puntos que no son válidos para el caso de las CPU multi-core. Asumimos que las CPU convencionales son un recurso único e indivisible; y que si garantizamos que los procesos tienen iguales *slices* de *quantum*, entonces comparten la CPU de forma justa o igualitaria. Esta aproximación no siempre es válida para CPUs multi-core, mayoritariamente por la influencia de la cache [36], y el problema es perceptible para los procesos en ejecución en una CPU multi-core si la L2 es compartida. Destacamos que es bastante común en los diseños de CPUs multi-core que la L2 sea compartida y los algoritmos de control y búsqueda en ellas estén a nivel de hardware [29, 57].

El problema creado al compartir dos *threads* una CPU multi-core es conocido como *variabilidad de rendimiento para procesos en co-ejecución dependientes*. Los procesos en *co-ejecución* son aquellos procesos que están en ejecución concurrentemente en la CPU multi-core, siendo bastante común que la cache se comparta de forma no justa, en dependencia de las necesidades de cache de cada proceso. El problema del *rendimiento variable* radica precisamente de esta circunstancia, la cantidad de cache ocupada por un proceso define las veces que falla la cache, y en consecuencia la velocidad a la que puede retirar instrucciones, con un impacto directo en el rendimiento.

Esta situación tiene varias consecuencias, como lo son:

- La CPU se comparte de forma no equitativa: la influencia de la cache hace que aunque los planificadores le asignen los mismos intervalos de

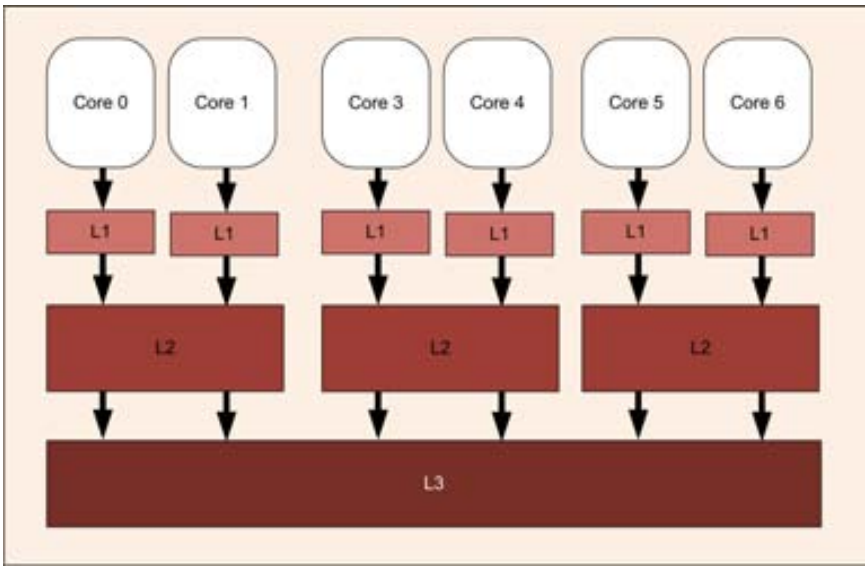

Figura 1.11: Arquitecturas de cache de procesadores multi-core, L2 parcialmente compartida y L3 compartida.

tiempo y ejecuten la misma cantidad de instrucciones, el tiempo de ejecución se vea afectado por los fallos de la cache.

- Deficiencias en el manejo de las prioridades: aún cuando un proceso tenga más prioridad, si los procesos en co-ejecución ocupan la cache este aumento en la prioridad no se verá reflejado en el tiempo de ejecución final.
- Insuficiencias a la hora de contabilizar el tiempo de CPU: el tiempo real que un proceso consume varía en dependencia de los procesos en co-ejecución.

Esta situación es compleja, pero en nuestro caso este problema se ve minimizado por la baja contención de recursos que encontramos en los laboratorios de universidades, resultado que también encontramos en [16]. En dicho trabajo se exponen los resultados de un muestreo realizado en los laboratorios de estudiantes de la Universidad Simon Fraser, localizada en Burnaby, Canadá. Los datos arrojan que durante cerca del 90 % del tiempo la longitud de la Ready Queue de los ordenadores muestreados ha sido de 0 ó 1. Esta información corrobora nuestro estudios, permitiéndonos simplificar nuestro escenario de trabajo, al considerar que para nuestro escenario particular existe una baja contención de los recursos, debido a la longitud de la Ready Queue de las CPUs. Otro punto a nuestro favor es que conocemos de antemano toda la carga a lanzar en los nodos, tanto de los usuarios locales como de los paralelos, inclusive los posibles tipos de las aplicaciones.

1.4.2. Características distintivas

En esta sección mencionamos algunas de las características distintivas de los procesadores multi-core, relacionadas con los threads, la afinidad y la coplanificación.

1.4.2.1. Afinidad

Definimos la afinidad como la máscara identificativa del core en el cual se ejecutará un proceso determinado. Esta máscara puede contener un valor que no indique ningún índice de core determinado, situación en la cual el sistema operativo será libre de mover el proceso entre los cores del procesador. A partir de este momento nos referiremos indistintamente a la afinidad como afinidad del procesador o afinidad del core indistintamente, identificando el índice o los índices de los cores en los que se ejecuta la tarea o proceso al que nos estemos refiriendo.

Debido a que la implementación actual de Linux incluye una rutina de balanceo de colas que se ejecuta periódicamente, hemos encontrado que es común establecerle un valor determinado a la afinidad, con el fin de facilitar los procesos que involucran procesadores multi-core. En este trabajo optamos por emplear la afinidad en nuestras políticas de asignación de cores para reforzar los diferentes requerimientos de nuestro entorno de planificación, la planificación de diferentes tipos de aplicaciones y la protección de los usuarios locales. Dichas políticas son descritas en el capítulo 2.

1.4.2.2. Coplanificación de threads

Una opción interesante al hablar de procesadores multi-core es planificar a la vez threads, idea que constituye la evolución natural de la coplanificación de threads para procesadores SMT [91].Trabajo en el que demuestran que el rendimiento de los procesadores multithreads (SMT) es sensible a las aplicaciones que son coplanificadas en él. Afirman además que si un procesador es capaz de reconocer las interacciones entre los threads, aprovecha mejor su capacidad multithread.

Por otra parte en [58] se proponen políticas de particionamiento para planificar tareas SRT en procesadores SMT, empleando el factor de simbiosis de las tareas para mejorar los resultados. La simbiosis describe como se mezclan de manera adecuada las tareas en el procesador, y se calcula en base a caracterizaciones realizadas de los procesos en procesadores que no son SMT.

En **[42]** determinan la importancia de la coplanificación en entornos multicore para el rendimiento de aplicaciones paralelas a través de experimentación. Crean una serie de recomendaciones que abordan tres puntos princi-

pales, la clasificación de los procesos, la capacidad de adaptarse y detectar su tipo de forma automática y la necesidad de incrementar la cooperación entre los procesos. Las conclusiones más importantes son:

1. Es necesario desarrollar aplicaciones capaces de aprovechar el paralelismo existente en el hardware.

2. Lograr un mejor comportamiento de los planificadores de los SO ante la existencia de los procesadores multicore.

1.4.2.3. Consumo de energía

Con el creciente número de transistores integrados a los procesadores incrementado de forma dramática en los procesadores multi-core, el problema del consumo de energía gana relevancia en el mundo científico. Los enfoques que encontramos se basan en el principio básico y lógico de apagar los cores que no están en uso y en controlar el voltaje.

En [56] se crea un algoritmo de compilación que escala de forma dinámica el uso del voltaje en los procesadores, y en [90] crea un esquema estático de compilación encargado de apagar los cores que no estén en uso y establecer un equilibrio entre el tiempo de ejecución y el consumo, estableciendo un límite en cuanto puede crecer el tiempo de ejecución.

Hemos encontrado algunos trabajos muy interesantes relacionados con el ahorro de energía, por ejemplo, en [7] se determinan los recursos ociosos en base a la estimación de la carga de trabajo, que se realiza en base a la longitud de las colas y los fallos de cache. Los recursos ociosos son desconectados. Encontramos un trabajo similar en [109], donde se crea un sistema de feedback para crear dominios de procesadores y controlar parámetros como la frecuencia y voltaje.

Aún cuando ajustar el consumo de energía es un punto a considerar en el futuro, en este punto de la investigación no lo hemos tenido en cuenta por las siguientes razones:

- El número de threads que se crea en las aplicaciones locales consideradas y el número de cores utilizado son poco significativos en nuestro entorno de desarrollo.

- En caso de escalar hacia más procesadores, podríamos aplicar técnicas basadas en compilación para optimizar el uso de energía.

1.4.3. Estado del arte

Aunque el aumento de la capacidad de cómputo introducido por la presencia de varias unidades de procesamiento en los procesadores es altamente atractiva, impone a los investigadores desafíos interesantes y complejos. La aparición de los procesadores multi-core provocó un gran interés en la comunidad científica, en este trabajo solo revisamos el estado del arte relacionado a nuestras necesidades, las NOWs y las tareas SRT.

1.4.3.1. Multi-core en NOWs

En nuestro trabajo queremos aprovechar la capacidad de cómputo extra existente, con el objetivo de mantener técnicas de probada eficacia en entornos sin procesadores multi-core, como es el caso de la coplanificación. En esta sección mencionamos algunos de los entornos de planificación con soporte para procesadores multi-core, que demuestra el interés despertado por esta nueva característica de los procesadores.

SLURM [60] es un entorno de planificación *open-source* diseñando para clusters de tipo Linux de todos los tamaños. En el trabajo [18] se actualizan tantos los algoritmos de distribución de trabajos como la interfaz de usuario que permite lanzarlos en el entorno para que soporten procesadores multi-core. Los objetivos de este rediseño son minimizar la contención en la cache y la memoria principal, así como evaluar el equilibrio en la contención de recursos. En general SLURM ha introducido mejoras en su entorno para permitir administrar las características distintivas de los procesadores multi-core, al igual que SGE [44] en su versión 6.2 Update 5 [3].

En el caso de PBS [54] no hemos encontrado trabajos que describan como ha sido modificado para soportar procesadores multi-core. Cabe mencionar que en su versión profesional si considera que los nodos puedan ser multiprocesador. Otro administrador de recursos que hemos encontrado carece de soporte para procesadores multi-core es el Load Leveler [75].

Entre los trabajos más recientes para NOWs dedicadas, sin considerar tareas SRT o RT, encontramos el de [111]. En este trabajo se adaptan las políticas de planificación espacial y temporal a la presencia de procesadores multi-core en los nodos. Las política propuestas son heurísticas, debido a que la complejidad del escenario genera un gran número de posibles combinaciones entre los grupos de tareas y los procesadores, ya que ahora pueden tener varios cores. Entre las simplificaciones realizadas consideradas encontramos que no asignan tareas de una misma aplicación en el mismo nodo.

1.4.3.2. Multi-core y SRT

Aunque obviamente el incremento de la capacidad de cómputo de los procesadores introducida por la presencia de varios núcleos es positiva para la planificación de tareas SRT, los métodos a usar para aprovecharla no son nada triviales. Este tipo de problema, pero para tareas RT y ordenadores multi-procesador, fue abordado en [61] a través de políticas de balanceo y reservas. En trabajos más actuales [9] se resuelve el mismo problema para procesadores de varios núcleos a través de una modificación de la política Pfair [100], con la ventaja añadida de no afectar la cache compartida. El enfoque se basa en agrupar las tareas de acuerdo al tráfico que generan entre la memoria y la L2, para luego poder reducir la concurrencia en tiempo de ejecución.

En [24] encontramos una propuesta para planificar tareas RT en plataformas multicore a larga escala con caches jerárquicas compartidas. Aunque la asignación es estática, el tamaño de los clusters de núcleos creados se calcula en base a las características de los conjuntos de tareas RT. La propuesta explota la "agrupación natural de los núcleos" para crear los antes mencionados clusters de núcleos.

En el trabajo [78] se intenta solucionar el problema que supone hacer la asignación de las tareas a los procesadores y sus prioridades. Se limitan a tareas con tiempos de ejecución estocásticos, pues de esta cualidad nacen las ventajas del algoritmo, que emplea métodos estocásticos para evitar restricciones fuertes impuestas por la caracterización de las tareas. Por el tipo de propuesta, el método solo es válido para tareas SRT. La experimentación realizada demuestra que para los parámetros dados el método es más eficiente que trabajos basados en garantizar el peor tiempo de ejecución de la tarea SRT.

En relación a las tareas SRT y los procesadores multicore también encontramos el trabajo [8]. En este trabajo muestran que las aplicaciones RT con varios threads que son cooperativos y comparten un working set común, resulta ventajoso planificar los threads particulares a la vez. Para lograr esto crean grupos de tareas. La propuesta garantiza las restricciones tiempo real de dichas tareas y hace un uso óptimo de las caches compartidas.

1.5. Sistema de planificación para aplicaciones de múltiples tipos en entornos multi-core

Nos enfrentamos a una problemática compleja, hemos de considerar nuevos tipos de aplicaciones de características especiales y lanzarlas en un hardware

que en si mismo constituye un desafío. Para resolver este problema necesitamos una plataforma de experimentación flexible, que nos permita llevar a cabo nuestros estudios. Hemos estudiar la planificación temporal y espacial de aplicaciones con características SRT, tanto locales como paralelas, en entornos no dedicados con procesadores multi-core, manteniendo el soporte para la ejecución de aplicaciones Best-effort, que también podrán ser paralelas o locales.

Como resultado de trabajos previos, en nuestro grupo hemos desarrollado CISNE (*Cooperative & Integral Scheduling for Non-dedicated Environments*) [49, 52], una propuesta para la utilización de recursos no dedicados, que implementa una Máquina Virtual Paralela (MVP) y utiliza técnicas de planificación de aplicaciones. Este sistema proporciona una doble funcionalidad (Figura. 1.12): ejecutar aplicaciones paralelas de tipo Best-effort y aplicaciones locales (Best-effort), pertenecientes a los usuarios locales del cluster no dedicado.

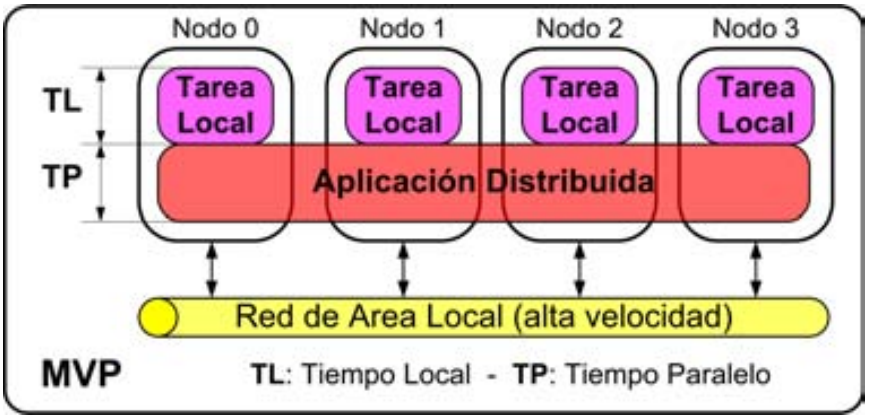

Figura 1.12: Máquina virtual paralela.

En su implementación inicial CISNE considera que cada nodo del cluster es compartido en el tiempo por ambos tipos de carga Best-effort: local y paralela. En consecuencia el sistema ha de gestionar el uso de los recursos entre las aplicaciones que se ejecutan, considerando que las tareas locales no pueden verse ralentizadas. Otro objetivo es garantizar el progreso de las tareas de las aplicaciones paralelas Best-effort en ejecución, de forma tal que el usuario local no note una intrusión en su ordenador. Esta propuesta se basa en una técnica de *Planificación de Aplicaciones Paralelas*, que considera las características de las aplicaciones distribuidas y el estado del entorno para ejecutar las aplicaciones de los usuarios paralelos.

Para aplicar la Planificación de Aplicaciones se analiza el problema desde dos dimensiones opuestas y complementarias: el espacio y el tiempo. Como se ha dicho, cada nodo de la MVP ha de ser capaz de gestionar el uso de CPU

entre las tareas en ejecución, aspecto que que se conoce como ***Planificación Temporal*** (P.T.). Desde el punto de vista del espacio, el sistema ha de ser capaz de asignar el conjunto de nodos que conforman el cluster no dedicado a las aplicaciones paralelas que los necesiten, garantizando que ningún nodo será sobrecargado de forma que las tareas locales vean alterada su capacidad de respuesta. Este tipo de planificación es conocida como ***Planificación Espacial*** (P.E.) y es el principal objetivo tomado en cuenta en el diseño original del sistema.

Para evaluar nuestras propuestas, hemos de *extender las funcionalidades de CISNE*, principalmente siguiendo las directivas enumeradas a continuación:

1. Los tipos de cargas de trabajos soportados inicialmente por el sistema (aplicaciones locales y paralelas de tipo Best-effort) han de ser ampliados para soportar nuestras necesidades de experimentación. Hemos de ser capaces de trabajar con aplicaciones con características SRT, tanto locales como paralelas, pues nuestro escenario de experimentación está compuesto por los cuatro tipos de carga de trabajo.

2. Ha de extenderse el diseño de CISNE para permitir un marco más flexible en la planificación temporal, ya que su idea inicial de diseño está enfocada en la planificación espacial.

3. Ser capaces de predecir, con niveles aceptables de precisión, los tiempos de turnaround de las aplicaciones paralelas. Cabe destacar que el nuevo escenario nos ha obligado a desarrollar nuevos métodos de estimación.

4. Hemos de dotar a nuestro entorno de planificación de funcionalidades relacionadas con la presencia de procesadores multi-core, como la posibilidad de establecer la afinidad de las tareas a determinados núcleos de los procesadores.

5. Hemos de facilitar la inclusión de políticas de planificación temporal, para su estudio y evaluación en nuestro sistema. Dichas políticas consideran tanto los nuevos tipos de aplicaciones SRT, como los procesadores multi-core

Recalcamos que al grupo inicial de aplicaciones previstas en el diseño de CISNE (locales y paralelas de tipo Best-effort), hemos de agregar los nuevos tipos de aplicaciones existentes en nuestro escenario. Finalmente, los tipos de aplicaciones con las que ha de ser capaz de trabajar el sistema para desarrollar nuestros estudios serían:

- *Locales Best-effort*: Tipo de aplicaciones locales "comunes", usualmente editores de texto, compiladores y aplicaciones con niveles de interactividad que pueden ser medidos con respecto a la respuesta en un

tiempo acotado por la capacidad de reacción del ser humano al utilizar el teclado o el ratón.

- *Paralelas Best-effort*: Aplicaciones paralelas para las cuales no existen limitaciones en el turnaround o exigencias de QoS. La principal cualidad deseada consistía en predecir lo mejor posible el tiemop de retorno, para lograr una mejor planificación y brindarle información al usuario paralelo.

- ***Locales SRT***: Aplicaciones con requerimientos de recursos determinados, usualmente necesitan la CPU de forma periódica para su correcta ejecución. Es también una buena idea garantizarles la cantidad de memoria principal que necesitan, ya que en caso contrario podrían no ejecutarse correctamente. Descritas en la sección 1.3.2.1.

- ***Paralelas SRT***: Aplicaciones paralelas con un turnaround determinado debido a sus características. Descritas en la sección 1.3.2.1.

Es válido destacar que aún cuando en la literatura encontramos trabajos que estudian el comportamiento de las aplicaciones paralelas SRT en clusters, nuestro trabajo se diferencia por el hecho de contemplar carga local, tanto de tipo Best-effort como SRT. En [112, 116] se estudia el comportamiento de aplicaciones SRT en clusters dedicados y [31, 103] describen herramientas dedicadas a la planificación espacial de tareas paralelas SRT. En [31] los nodos se seleccionan de acuerdo a estudios probabilísticos y en [103] se intenta garantizar los deadlines de las tareas creando las tareas como parejas. Este enfoque permite tener dos niveles de prioridad, para poder lanzar la tarea con más prioridad si se estima que perderá su deadline.

Por otro lado, estudios como [4] intentan garantizar un recurso crítico, en este caso el ancho de banda de red, empleando mecanismos de QoS. En [113] también se hace uso de mecanismos de QoS para intentar garantizarles recursos a algunos tipos de aplicaciones, la novedad es que particionan los recursos disponibles para lograr hacerlo.

Una vez introducido el estado del arte y los conceptos básicos para comprender el alcance y enfoque de nuestro trabajo, pasamos a introducir los objetivos que nos proponemos.

1.6. Objetivos

El nacimiento de nuevos tipos de aplicaciones (locales y paralelas de tipo SRT) impone estudiar las técnicas de planificación espacial y temporal para lograr la coexistencia de diferentes tipos de aplicaciones en clusters no dedicados. Hemos encontrado diferentes tipos de aplicaciones paralelas, como

las científicas, de procesamiento de imágenes o sencillamente con un deadline impuesto, que conforman un nuevo tipo de aplicaciones paralelas que necesitan una atención especial por parte de los sistemas de planificación. Por otro lado, muchos estudios se enfocan en la evolución de las aplicaciones locales, la forma de identificarlas y de garantizarles los recursos necesarios. Uno de nuestros objetivos principales es, crear un sistema de planificación de aplicaciones paralelas para clusters no dedicados, que tenga en cuenta la importante evolución de las aplicaciones de los usuarios involucrados en este escenario.

En los últimos años se ha producido una evolución importante en los procesadores, la capacidad multicore. Esta situación han originado una gran atención en la comunidad científica, debido a los múltiples desafíos que representa. Las ventajas brindadas por este nuevo tipo de hardware presenta un problema adicional, pues para sacar maximizar su uso han de recrearse muchos de los principios establecidos en la actualidad, máxime si involucra tipos de aplicaciones con requerimientos soft-real time. De especial importancia para nosotros es la masificación de esta tendencia, que nos ha conducido a introducir modificaciones a nuestro prototipo con el objetivo de aprovechar eficientemente nuevo el potencial de cómputo.

Nuestro trabajo entrelaza varias áreas de investigación bien definidas: la planificación de aplicaciones paralelas, el tiempo real débil y el uso de procesadores multi-core. Nuestro trabajo combina dichas áreas de investigación, ***con el objetivo de incluir aspectos soft real-time en la planificación espacial y temporal en clusters no dedicados, tomando en cuenta que los nodos de las NOWs pueden tener procesadores multi-core***. La intersección de estas problemáticas crea un escenario altamente complejo, por lo que proponemos variantes a los modelos tradicionales, como lo es nuestro esquema de simulación a dos niveles.

El principal objetivo de este trabajo es proveer un sistema que permita la planificación de aplicaciones de varios tipos considerando la presencia de procesadores multi-core en las estaciones de trabajo, tanto para simulación como para ejecuciones reales. Los objetivos principales a conseguir, mediante la realización de este trabajo, son los que se presentan a continuación:

1. Crear e implementar políticas que permitan aprovechar el potencial de los procesadores ***multi-core*** para realizar la planificaciónt temporal de la mezcla de aplicaciones paralelas y locales, ya sean SRT o Best-effort. Las versiones más actuales de los sistemas operativos ofrecen cierta gama de facilidades para administrar las características distintivas de los procesadores multi-core. Pese a esta situación, la gran mayoría de los usuarios son incapaces de utilizarlas, abriendo una brecha entre el potencial de los procesadores y su aprovechamiento. En este punto hemos desarrollado dos políticas de asignación de cores: ***BY_APP***,

diseñada para mejorar el rendimiento del sistema para aplicaciones SRT; y ***BY_ USR***, diseñada para proteger al usario local en un escenario más agresivo.

2. Migrar el entorno de planificación usado como punto de partida en nuestro trabajo, denominado CISNE, de la versión 2.4 a la 2.6 del kernel de Linux.

3. Implementar en CISNE las modificaciones necesarias para que considerase los nuevos tipos de aplicaciones y la presencia de procesadores multi-core. Hemos trabajado en los módulos de CISNE relacionados con la planificación temporal para los nuevos tipos de aplicaciones SRT y el multi-core; y en el planificador espacial para implementar un sistema de adimisión de peticiones.

4. Implementar nuevos núcleos de simulación para CISNE que consideren la problemática actual, esta parte del trabajo se desarrolló en dos etapas fundamentales:

   a) Inclusión de un núcleo de estimación analítico para evaluar nuestras intuiciones a efecto de implementar nuevas políticas de planificación.

   b) Creación de un núcleo de estimación basado en simulación, opción que ha probado ser la que mejores resultados brinda debido a el incremento en la complejidad de nuestro escenario. Los esfuerzos por mejorar nuestro método de simulación, han dado origen a un nuevo núcleo tipo de núcleo de estimación también basado en simulación. Este núcleo de estimación simulado es capaz de procesar los tipos de aplicaciones, paralelas y locales de tipo SRT y Best-effort, considerando que los nodos de la NOW tengan procesadores multi-core.

5. Comparar las prestaciones del entorno desarrollado con otros entornos tradicionales tipo PBS.

6. La coplanificación es una técnica importante en la planificación de aplicaciones paralelas, generando mejoras en el rendimiento en cualquiera de sus variantes. En este trabajo constataremos la validez de la coplanificación de tareas de aplicaciones, SRT o Best-effort, para procesadores multi-core. Cabe destacar un punto importante de nuestro trabajo, está implementado totalmente en espacio de usuario.
   Nuestro enfoque se basa en estimular la coplanificación, basándonos en el control del grado de multiprogramación y la selección de las aplicaciones paralelas de acuerdo a su uso de los procesadores de los nodos, que puede ser de cómputo intensivo (CPU bound) o comunicación (IO

bound). Estas heurísticas son conjugadas con el sistema de prioridades interno de nuestro planificador para lograr nuestro objetivo: incrementar las probabilidades de que ocurra la coplanificación.

1.7. Organización de la Memoria

En este trabajo nos centramos en la planificación temporal de aplicaciones de diferentes tipos en clusters no dedicados, considerando además que las estaciones de trabajo tengan procesadores multi-core. Para lograr nuestro propósito creamos políticas encaminadas a potenciar el comportamiento del entorno de acuerdo a nuestros objetivos. Para probar el funcionamiento de estas políticas, ha sido necesario dotar a CISNE, un entorno de planificación desarrollado en nuestro grupo, de capacidades para ajustarse a los cambios ocurridos en este escenario. La presente memoria de organiza en los siguientes capítulos:

- *Capítulo 1*: Recoge el marco teórico de este trabajo, con secciones dedicadas a cada una de las áreas de investigación abordadas:
  - Las secciones 1.1 y 1.2 detallan los elementos relacionados con la planificación en NOWs, necesarios para ubicar y comprender nuestro trabajo.
  - La sección 1.3 introduce la terminología relacionada con el tiempo real, y lo que es más importante para nosotros, el tiempo real débil o soft real-time.
  - La sección 1.4 contiene los puntos relacionados con los procesadores multi-core tomados en cuenta en este trabajo.
- *Capítulo 2*: se detallan tanto las propuestas de planificación temporal basadas en la asignación de cores como las características particulares del entorno real contra el que nos comparamos y en el que realizamos nuestra experimentación.
- *Capítulo 3* describe nuestro esquema de simulación y las potencialidades que nos brinda.
- *Capítulo 4* introduce nuestro entorno de planificación para las ejecuciones reales.
- *Capítulo 5* describe los detalles de la implementación tanto de nuestro simulador como del entorno para ejecuciones reales, incluye el diseño de clases y relaciones entre ellas.

- *Capítulo 6* contiene los resultados experimentales que dan validez a nuestras propuestas, tanto para la simulación como para las ejecuciones reales.
- *Capítulo 7* presenta las conclusiones de este trabajo, junto con las contribuciones realizadas a lo largo del mismo. También incluye las principales líneas abiertas propuestas tomando en cuenta los resultados obtenidos y el estado de la investigación.

Estos capítulos se completan con los siguientes apéndices:

- *Apéndice* A incluye los puntos a tener en cuenta a la hora de lanzar aplicaciones en nuestro entorno de planificación.
- *Apéndice* B describe la forma de uso de la herramienta de simulación resultado de este trabajo.

Capítulo 2

Consideraciones y Propuestas

En este capítulo describiremos el entorno real con el que contamos para realizar nuestra experimentación, las políticas propuestas y algunas consideraciones relacionadas con nuestro trabajo, el Contrato Social y el entorno de planificación PBS.

2.1. Caso de estudio

En esta sección describiremos en detalle la configuración del hardware existente en nuestro cluster de pruebas, así como las particularidades del sistema operativo instalado en las estaciones de trabajo.

2.1.1. Arquitectura de hardware

La arquitectura de procesador multi-core accesible en este trabajo crea un límite natural para nuestros esfuerzos. Como se menciona en el capítulo de experimentación, las estaciones de trabajo de nuestro cluster tienen un procesador Intel® Pentium® D Processor 950 [1] a 3.40 GHz, cuyas especificaciones mostramos en la Tabla 2.1.

sSpec Number	SL95V	*Package Type*	LGA775
CPU Speed	3.40 GHz	*Manufacturing Technology*	65 nm
Bus Speed	800 MHz	*Core Stepping*	C1
Bus/Core Ratio	17.0	*Thermal Design Power*	95W
L2 Cache Size	4 MB (2x2)	*Thermal Specification*	63.4°C

Tabla 2.1: Especificaciones del procesador Intel® Pentium® D Processor 950.

De esta tabla queremos destacar los datos relacionados con la cache L2, compuesta de un módulo de 2 MB para cada core que tiene el procesador. Problemas como la co-ejecución de procesos y sus consecuencias vienen dados por el uso de cache compartidas, problemas en nuestro caso de estudio no encontraremos debido a la arquitectura de cache existente. La combinación de esta arquitectura de cache con el conocimiento de nuestro escenario, colas de ejecución con pocos procesos y conocimientos de las aplicaciones, nos permite minimizar los problemas de contención de recursos a través del balanceo por el bound de las aplicaciones paralelas.

Las características de este procesador son las siguientes:

- *Dual Core*: el procesador incorpora dos unidades de ejecución o cores en un solo chip.
- *Enhanced Intel Speedstep® Technology*: permite al sistema ajustar dinámicamente el voltaje del procesador y la frecuencia del núcleo. Estos ajustes pueden conducir a la disminución de los promedios en el consumo de energía y la generación de calor.
- *Intel® EM64T*: tecnología de Intel que permite direccionamiento con 64 bit.
- *Intel® Virtualization Technology*: tecnología de Intel para la virtualización de los recursos físicos de un sistema informático, con el objetivo de mejorar utilización y capacidad de compartir.
- *Enhanced Halt State* (C1E): tecnología para un menor consumo de energía.
- *Execute Disable Bit*: funcionalidad que puede ayudar a prevenir ataques maliciosos de desbordamiento de buffer cuando se combina con un sistema operativo compatible.

Este procesador constituye una de las últimas versiones de esta familia de procesadores.

2.1.2. Núcleo de Linux versión 2.6

Entre las variadas mejoras introducidas con la entrega del kernel 2.6 de Linux, contaremos con el mayor nivel de detalle posible las relacionadas con el *proceso de planificación*. Estas mejoras incluyen:

1. Cambios en la forma de clasificación de los procesos.
2. Mejoras en la apropiatividad.

3. Algoritmo de planificación en tiempo constante O(1).

El proceso de planificación puede ser contado en dos partes, una relacionada con las abstracciones introducidas y la segunda relacionada con el propio algoritmo.

Política de planificación

La parte del proceso de planificación conformada por un conjunto de reglas para determinar cual será el próximo proceso en ejecutarse se denomina *política de planificación*. La planificación en Linux está basada en compartir el tiempo, lo cual significa que los procesos ejecutables disponen de *slices* o *quantums* del tiempo de la CPU, y que cuando este quanto de tiempo expira, si el proceso aún no ha terminado, ocurrirá un intercambio de procesos. Esta técnica está basada en interrupciones del reloj interno o *timer*, y es transparente a los procesos.

Otro de los factores tomados en cuenta por la política de planificación es la *prioridad* de los procesos, que indica cuan apropiado es permitir a un proceso ejecutarse en la CPU. En Linux las prioridades de los procesos son dinámicas, el planificador recopila información de la actividad de los procesos y en base a esa información ajusta las prioridades. Esta técnica permite que los procesos que no han recibido la CPU por un período largo puedan incrementar su prioridad, y viceversa.

En la versión 2.6 se emplea una nueva forma de clasificar los procesos, que contempla tres clases:

1. *Interactivos*: Son los procesos que interactúan de forma constante con el usuario, involucrando gran cantidad de acciones por parte del mismo, como operaciones de ratón o teclado. Este tipo de procesos gasta mucho tiempo de CPU esperando por estos eventos, y necesita que al ocurrir un evento se le asigne la CPU rápidamente, en caso contrario el usuario podría encontrar que el sistema no le responde. Los ejemplos más típicos son los editores de texto, aplicaciones gráficas o líneas de comandos.

2. *Batch*: No necesitan interactuar con el usuario, por lo que usualmente se ejecutan en segundo plano. Debido a esta característica son los procesos que el planificador penaliza con más frecuencia. Ejemplos de este tipo de trabajo son los compiladores o los motores de búsqueda en bases de datos.

3. *Tiempo real*: Este tipo de procesos tiene características muy particulares, mencionadas en las sección 1.3.

Antes de continuar con la política de planificación empleada en el kernel versión 2.6 queremos destacar que aunque teóricamente permite ejecutar procesos tiempo real, en la práctica para nuestro caso particular el enfoque empleado hace que tengamos que desestimar la abstracción creada con esta finalidad. La razón principal es que los procesos que se colocan en la cola de tiempo real de Linux se ejecutan mientras tengan necesidad de la CPU, penalizando gravemente al resto de los procesos. Como para nuestro caso particular reviste especial importancia que las aplicaciones de los usuarios locales mantengan los niveles de respuesta, tenemos que descartar en emplear las colas tiempo real incluidas en la versión 2.6 del kernel de Linux. Otra desventaja seria es que los procesos que se colocan en esta cola no tienen noción de período, tiempo de cómputo o deadline; que como mostramos en la sección 1.3 son imprescindibles a la hora de emplear cualquiera de los algoritmos de planificación para tiempo real existentes.

Cabe mencionar que aunque la documentación del kernel de Linux menciona que el mismo implementa un sofisticado algoritmo para diferenciar los procesos interactivos de los batch, en trabajos como [35] se demuestra que la diferenciación es un proceso complejo, en el que se incurren en muchos falsos positivos.

Algoritmo de planificación

En versiones previas a la 2.6 el algoritmo de planificación de Linux escalaba mal para grandes cantidades de procesos, debido a que cada vez que hacía un cambio de proceso tenía que recorrer toda la lista de procesos ejecutables, calcular sus prioridades y en base a esta información seleccionar el mejor proceso para ser ejecutado. El algoritmo de planificación introducido con la versión 2.6 del kernel no tiene esta desventaja, pues selecciona el mejor proceso para ejecutar empleando un tiempo constante, independiente de la cantidad de procesos ejecutables. Además, como cada core (o CPU para multiprocesadores) tiene su propia cola de procesos, escala bien para varios cores o procesadores.

En esta versión del kernel existen tres *clases de planificación*, las cuales son:

1. SCHED_FIFO: clase de cola para tiempo real con el paradigma First-In First-Out. Cuando el planificador asigna la CPU a un proceso tiempo real, el proceso deja su *descriptor* en su posición actual en la *runqueue*. Si no existe otro proceso tiempo real con mayor prioridad en estado ejecutable, el proceso que ha recibido la CPU se ejecuta durante el tiempo que necesite, aún si existen otros procesos con la misma prioridad en estado ejecutable.

2. SCHED_RR: clase de cola para tiempo real con el paradigma Roud Robin. Cuando el planificador asigna la CPU a un proceso tiempo real,

el proceso deja su *descriptor* en su posición final en la *runqueue*. Esta política garantiza que la CPU se comparta de forma justa entre todos los procesos tiempo real con la misma prioridad que estén en esta cola.

3. SCHED_NORMAL: clase de cola convencional de Linux, de tiempo compartido.

Como ya mencionamos antes, las colas tiempo de real que implementa Linux carecen de interés para nuestro trabajo, por las grandes penalizaciones que introducen a los demás procesos. Pasamos a explicar el comportamiento de la cola convencional de Linux, de tiempo compartido, que será la empleada para ejecutar tanto nuestro planificador como las aplicaciones que consideramos.

Cada proceso tiene una prioridad estática propia, que constituye el valor con el que el planificador compara el proceso con los otros procesos convencionales del sistema. Los valores de la prioridad estática pueden variar entre 100 y 139, siendo 100 el valor más prioritario y 139 en menos prioritario, nótese que la prioridad decrece al incrementarse el valor.

Un nuevo proceso siempre hereda su prioridad del proceso padre, aunque los usuarios pueden variar la prioridad de sus procesos a través de los valores "*nice*" pasados como parámetros a las funciones *nice()* y *setpriority()*. Este valor se emplea para determinar el tiempo básico del quantum que recibirá el proceso luego de agotar su quantum actual. La Ecuación 2.1 muestra la relación entre la prioridad y el tiempo básico del quantum [76].

$$btq = \begin{cases} (140 - staticprio) \times 20 & if\ staticprio < 120 \\ (140 - staticprio) \times 5 & if\ staticprio \geq 120 \end{cases} \tag{2.1}$$

Como se desprende de esta ecuación, a las mayores prioridades estáticas, que se representan con los menores valores numéricos, les corresponden los mayores valores de quantum. Para ilustrar mejor la relación entre estos valores hemos incluido la Tabla 2.2.

Prioridad estática	Valor "nice"	Tiempo base del quantum	Descripción
100	-20	800 ms	Prioridad estática más alta
110	-10	600 ms	Prioridad estática alta
120	0	100 ms	Prioridad estática por defecto
130	+10	50 ms	Prioridad estática baja
139	+19	5 ms	Prioridad estática más baja

Tabla 2.2: Relación entre la prioridad y el quantum base.

Además de un valor de prioridad estática, los procesos convencionales o no tiempo real en Linux también tienen asociada una *prioridad dinámica*, cuyo valor está comprendido entre 100 (la más alta) y 139 (la más baja). Este valor es el que utiliza el planificador de Linux para seleccionar cual será el próximo proceso en ejecutarse, y se calcula a través de la Ecuación 2.2.

$$dyn\,prio = max(100,\, min(static\,prio - bonus + 5,\, 139)) \tag{2.2}$$

En esta ecuación *bonus* representa un valor que oscila entre 0 y 10, si el valor de bonus es menor que 5 es una penalización que disminuye la prioridad dinámica; en cambio si es mayor que 5 la aumenta. El valor del bonus se calcula de acuerdo al tiempo promedio que pasa el proceso en estado *sleep*, que nunca será mayor que 1 segundo. Y es precisamente a través del valor del bonus que Linux decide si un proceso es interactivo o batch, empleando la Ecuación 2.3. En esta ecuación el valor $static\,prio/4 - 28$ es conocido como el delta interactivo.

$$bonus - 5 \geq static\,prio/4 - 28 \tag{2.3}$$

Como resultado de este procedimiento para calcular las prioridades dinámicas y determinar las aplicaciones interactivas resulta que es mucho más fácil para un proceso con prioridad alta convertirse en interactivo. Por ejemplo, un proceso con prioridad estática de 100, la máxima posible, pasa a ser considerado interactivo cuando su valor de bonus es mayor que 2, lo que viene a ser lo mismo que tener un tiempo promedio en sleep mayor que 200 ms.

Todo este complejo proceso para "crear" una forma dinámica de asignar las prioridades, y por ende el orden en el que los procesos entran en la CPU, se inspira en las siguientes heurísticas:

- La experiencia y la lógica nos llevan a intentar emplear una longitud del quantum que no sea ni demasiado largo ni demasiado corto:
  - Si el quantum fuese demasiado corto crearía overhead por los cambios constantes de procesos
  - Si el quantum fuese demasiado podría tener implicaciones con el nivel de respuesta del sistema a los eventos del usuario local, además de afectar la concurrencia de los procesos
- Linux intenta crear un compromiso, elige la mayor duración posible que no afecte el tiempo de respuesta del sistema operativo

Cuan exitoso es este procedimiento para lograr sus objetivos es un tema subjetivo, que depende tanto del usuario frente al sistema como de las aplicaciones que se lanzan, de la capacidad de Linux para adivinar correctamente cuando una aplicación es interactiva o no y más factores aún.

Finalmente mencionar que cada planificador mantiene dos conjuntos disjuntos de procesos ejecutables para evitar que los procesos con mayor prioridad consuman toda la CPU. Estos conjuntos son los siguientes:

Procesos activos: los procesos ejecutables que aún no han agotado su quantum.

Procesos expirados: los procesos ejecutables que han agotado su quantum, tienen que esperar a que todos los procesos activos agoten su quantum para tener permiso para ejecutarse.

Balanceo de colas de ejecución

La implementación del kernel 2.6.x de Linux se adhiere al modelo SMP, por esta razón el planificador no ha de tener predilección especial por ninguno de las CPU con respecto a las otras. Para lograr este objetivo, el planificador ha de ser capaz de comportarse diferente para cada configuración de hardware. La implementación actual es capaz de reconocer las siguientes arquitecturas:

- Clásica: memoria principal compartida entre todas las CPUs
- Hyper-threading: es un microprocesador que ejecuta varios threads a la vez, Linux implementa una CPU con hyper-threading como varias CPUs lógicas diferentes
- NUMA: arquitectura de memoria que agrupa las CPUs con cierta cantidad de memoria principal, y emplea un circuito especializado para controlar los accesos, que son muy rápidos si accede los módulos locales de memoria y mucho más lento si ha de acceder a un módulo de memoria principal "perteneciente" a otra CPU.

Para evitar que varios procesos que hacen un uso pesado de la CPU terminen en la misma cola de ejecución Linux, a partir de la versión 2.6.7 del kernel, implementa un algoritmo de balanceo de colas basado en la noción de los "dominios de planificación". Este algoritmo también es el que se emplea para el balanceo en los procesadores multi-core.

En esencia un "dominio de planificación" es un conjunto de CPU cuya carga de trabajo ha de mantenerse balanceada. La organización es jerárquica, lo que permite un balanceo más eficiente. Los dominios a su vez se descomponen en grupos, cada uno de los cuales representa un subconjunto del dominio. El

balanceo de cargas se realiza siempre entre grupos de un dominio de planificación, lo cual significa que una tarea se mueve de una CPU a otra solamente si alguno de los grupos de un dominio de planificación tiene una carga significativamente más baja que el otro grupo dentro del mismo dominio.

La simplificación de este algoritmo para procesadores dual-core, como los empleados en este trabajo, implica que cada aproximadamente 200ms se ejecuta la rutina de balanceo y que si uno de los dos cores está más cargado que otro se migra la tarea. Esta situación puede generar mezclas de threads que no interesan en un entorno paralelo.

Aunque la versión en uso del kernel de Linux en este trabajo es la 2.6.7, creemos importante contar algunas de las mejoras introducidas en la versión 2.6.18 por Ingo Molnar para mejorar las prestaciones de Linux ante tareas RT.

- mejoras en la apropiatividad del kernel, si bien aún no es completamente apropiativo, se ha mejorado este aspecto
- hrtimer: subsistema para proveer un timer de alta resolución
- planificador por clases: sin dependencias laterales

2.2. Contrato Social

El concepto del Contrato Social, introducido en [12], establece un límite entre los recursos disponibles para los usuarios locales y paralelos en clusters no dedicados. Este límite permite al planificador espacial inyectar aplicaciones paralelas en las estaciones de trabajo de las NOWs sin sobrecargarlos, manteniendo de esta forma el nivel de respuesta del ordenador necesario para los usuarios locales.

El contrato social se basa en establecer un límite en los recursos que pueden emplear las aplicaciones paralelas, que en el momento actual contempla los dos recursos que hemos identificado como críticos, la memoria principal y la CPU. Para que este factor tenga sentido han de tomarse en cuenta los siguientes puntos:

- Características de hardware de las estaciones de trabajo que conforman la NOW.
- Caracterización de las aplicaciones empleadas por los usuarios locales.

A partir de estos datos podemos establecer un límite en los recursos que podrá usar nuestro entorno de planificación para las aplicaciones paralelas. En

su concepción inicial este valor fijaba tanto la CPU como la memoria principal. En nuestro trabajo hemos fijado un valor fijo para el uso de la memoria principal, que tomando en cuenta la configuración de hardware de nuestro cluster de pruebas y la caracterización de las aplicaciones locales hemos fijado este valor en $L_{MP} = 0{,}85$. Para establecer el límite para el recurso CPU consideramos las características multi-core de los procesadores y las políticas que hemos definido para su uso. La forma en la que lo establecemos depende de cada política.

2.3. PBS

Este trabajo incluye una comparación entre nuestro entorno de planificación y PBS [21]. PBS es un administrador de recursos para Linux desarrollado por la NASA a mediado de los años 90. En la actualidad cuenta con una versión *open-source* denominada openPBS [19] y otras versiones comerciales mantenidas por diferentes empresas.

El diseño de PBS contempla la posibilidad de definir un módulo de planificación externo, aunque en la instalación empleada durante las ejecuciones empleamos el planificador por defecto. El planificador por defecto de PBS permite ordenar los trabajos tanto por colas FIFO como por criterios más sofisticados como las prioridades de usuarios y grupos, fairshare y desalojo. La configuración por defecto planifica tomando en consideración los siguientes puntos:

- Ordenamiento de las colas en orden descendente de prioridad para determinar el orden en que serán consideradas.
- Todos los trabajos en la cola de mayor prioridad serán ejecutados antes de pasar a la cola siguiente.
- Los trabajos son organizados en sus colas en orden creciente de acuerdo a la CPU que han solicitado.
- Se establecen límites temporales para descubrir trabajos que pasan mucho tiempo en las colas, marcándolos como *starving* e intentando ejecutarlos.
- Se considera una serie de parámetros a la hora de realizar la planificación, como lo son el tamaño de las colas, las cantidades máximas de trabajos y recursos por usuario y grupo, el estado del trabajo, la cola en la que se encuentre y si tiene necesidades específicas de arquitectura o nodo.
- Chequeo de los recursos del sistema, memoria y CPU, para garantizar que no se sobrepasen.

Para nuestro caso hemos introducido los siguientes parámetros introducidos a través del gestor *qmgr* de PBS:

 # Create and define queue workq
create queue workq
set queue workq queue_ type = Execution
set queue workq enabled = True
set queue workq started = True
#
 # Set server attributes.
set server scheduling = True
set server default_ queue = workq
..
set server query_ other_ jobs = True
set server resources_ default.neednodes = 8
set server resources_ default.nodect = 8
set server resources_ default.nodes = 8
set server scheduler_ iteration = 600
set server node_ pack = False

Al introducir estos parámetros creamos una cola denominada *workq*, de tipo ejecución y la activamos e iniciamos. Luego se configuran ciertos valores necesarios para el servidor.

Por otra parte el fichero de nodos en el servidor contiene los 8 nodos empleados durante la experimentación:

[root@aopcso1 PBS]# cat server_ priv/nodes
aopcso1 np=2
...
aopcso8 np=2

Nótese que se especifica para cada nodos la cantidad de procesadores que tiene.

Los detalles de la configuración del planificador de PBS, extraídos del fichero */var/spool/PBS/sched_ priv/sched_ config* en el nodo servidor son los siguientes:

round_robin: False all

strict_fifo: false ALL

fair_share: false ALL

help_starving_jobs true ALL

sort_queues true ALL

load_balancing: false ALL

sort_by: shortest_job_first ALL

max_starve: 24:00:00

Esta es la configuración por defecto del planificador de PBS, que significa que no se hará round robin entre las colas existentes, se vigilarán los trabajos para controlar que no estén en estado starving por más de 24 horas, no existe balanceo de cargas y el ordenamiento en las colas es por el trabajo más corto.

2.4. Políticas de asignación de cores

Una vez descritas en detalle tanto la configuración de hardware como las peculiaridades del kernel de Linux en uso, pasamos a describir las políticas propuestas en este trabajo.

Una de las características distintivas introducidas por los procesadores multi-core es la afinidad. Como ya hemos mencionado previamente, nos permite establecer una bandera empleada por el sistema operativo para decidir en que core del procesador se podrá ejecutar la tarea. En caso de contener un solo valor dicha tarea se verá excluida de las migraciones producidas por la rutina de balanceo del planificador de Linux.

En este trabajo empleamos la afinidad para crear políticas de asignación de tareas a cores que refuercen los objetivos perseguidos. Las políticas creadas se basan en los siguientes objetivos:

1. Proteger al usuario local de la invasión representada por la presencia de aplicaciones paralelas en sus estaciones de trabajo.

2. Garantizar suficientes recursos para los nuevos tipos de aplicaciones soft real-time, sin importar si son locales o paralelas.

Nuestras políticas han de ser capaces de planificar la mezcla de aplicaciones locales y paralelas involucradas en nuestro escenario, dicha mezcla se compone de la siguiente forma:

- *Locales Best-effort*: son las aplicaciones locales "comunes", usualmente editores de texto, compiladores y aplicaciones con niveles de interactividad que pueden ser medidos con respecto a la respuesta en un tiempo acotado por la capacidad de reacción del ser humano al utilizar el teclado o el ratón.
- *Paralelas Best-effort*: aplicaciones paralelas para las cuales no existen limitaciones en el turnaround o exigencias de QoS. La principal cualidad deseada consistía en predecir lo mejor posible el turnaround para lograr una mejor planificación y brindarle información al usuario paralelo.
- ***Locales SRT***: aplicaciones con requerimientos de recursos determinados, usualmente necesitan la CPU de forma periódica para su correcta ejecución. También es necesario garantizarles la cantidad de memoria principal que necesitan, ya que en caso contrario podrían no ejecutarse correctamente.
- ***Paralelas SRT***: Aplicaciones paralelas con requerimientos de QoS o un turnaround determinado.

Vale la pena destacar que cada política implementa el Contrato Social de acuerdo al uso que hace de los cores de la CPU.

Nuestras políticas comparten tanto las particularidades de nuestro escenario como las limitaciones que tenemos a la hora de realizar la experimentación. En este trabajo el cluster no dedicado que consideramos como escenario de trabajo son los laboratorios universitarios, donde los tipos de aplicaciones empleadas son conocidas, situación que nos permite asumir una serie de simplificaciones. Existe baja contención de los recursos, debido a los tipos y la cantidad de aplicaciones. Las aplicaciones lanzan pocos threads y las colas de ejecución son cortas. Por otra parte la limitación más notable para nuestro trabajo, en relación con la experimentación con aplicaciones reales, es que solamente contamos con procesadores de dos cores.

Con el fin de establecer un punto de comparación para las políticas de asignación de cores se ha creado una opción especial de nuestro entorno, que ejecuta nuestro planificador, pero no emplea la afinidad para asociar las tareas con algún core. Es válido destacar que esta opción, que hemos denominado **LIN**++ en el Capítulo de Experimentación Realizada y Resultados Obtenidos, se beneficia de las potencialidades de nuestro entorno, tales como el empleo de políticas de planificación espacial y control del MPL.

2.4.1. BY_APP

Como ya mencionamos antes, cada una de nuestras políticas persigue una finalidad específica. En este caso, como se desprende del nombre de la políti-

ca, que significa textualmente "por aplicación", dicha finalidad es facilitar el proceso de planificación temporal en las estaciones de trabajo en base al tipo de las aplicaciones. La clasificación de las aplicaciones para esta política se basa en su tipo, que puede ser SRT o Best-effort.

Teniendo en cuenta que diseñamos políticas para procesadores dual-core, con baja contención y colas de ejecución cortas, hemos creado una política de asignación de tareas a cores basada en el tipo de las aplicaciones. Esto se traduce en que dedicamos un core para cada tipo de aplicación, sin importar si la lanza el usuario local o el paralelo, como se muestra en la Figura 2.1.

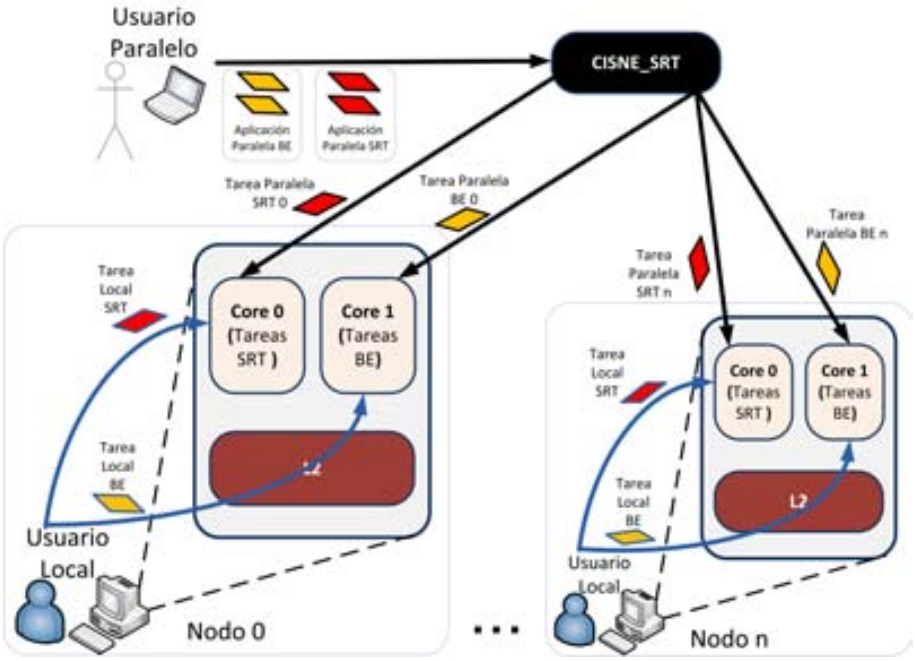

Figura 2.1: Política de asignación de cores por tipo de aplicación, denominada *BY_APP*.

En esta figura los tipos de aplicaciones están representados por colores diferentes para facilitar su identificación. Las flechas azules representan los lugares de destino de las tareas de los usuarios locales y las negras el movimiento de las aplicaciones paralelas y sus tareas en el cluster. Como resultado de la aplicación de esta política, las aplicaciones de tipo Best-effort competirán libremente entre ellas por el recurso CPU en uno de los cores del nodo, y podremos aplicar con mayor facilidad los algoritmos de planificación para tiempo real en el otro.

La combinación de esta política con el sistema de prioridades interno de nuestro planificador, descrito en la sección 4.2.4, y con la aplicación del Contrato Social; tiene los siguientes resultados:

- *Core dedicado a las tareas SRT*: el análisis de viabilidad se facilita enormemente, pues podemos aplicar directamente las ecuaciones creadas para cada grupo de algoritmos. También se facilita la implementación de los algoritmos tiempo real, pues podemos aplicar los resultados conocidos para procesadores que no son multi-core. Cabe destacar que nuestras políticas no consideran la migración de tareas entre los cores, al menos en esta etapa de nuestro trabajo.

- *Core dedicado a las tareas Best-effort*: basta con aplicar directamente en el Contrato Social, con un límite de recurso de CPU (L_{CPU}) fijado en $L_{CPU} = 0{,}85$, y dejar competir a las tareas dentro de los límites que este impone.

Básicamente el aplicar esta política hace viable considerar, dentro de un límite razonable, cada core como un procesador separado, y por lo tanto poder aplicar las soluciones encontradas para cada tipo de problema para procesadores que no son multi-core. Las diferencias entre emplear la política *BY_APP* y dejar trabajar solamente al planificador de Linux son las siguientes:

- Linux no es capaz de planificar aplicaciones SRT en sus colas RT sin degradar de forma ostensible el rendimiento del sistema para las demás aplicaciones. Si empleamos las colas RT para planificar el sistema puede perder capacidad de respuesta para el usuario, además de necesitar llamadas al sistema en modo superusuario para establecer este tipo de prioridad.

- Al crear nuestras políticas a través de un cambio en la afinidad de las tareas, inicialmente fijada en un valor que significa que la tarea puede estar en cualquier core del procesador, eliminamos la migración de tareas. Esta aproximación puede conducir a colas de ejecución desbalanceadas, aunque es el precio que hemos pagado por la simplificación del proceso de planificación lograda.

2.4.2. BY_USR

Para crear esta política nos hemos basado en otro de los objetivos principales de nuestro trabajo, si bien con un poco más de historia: el garantizar que los usuarios locales dispongan de recursos para lanzar sus aplicaciones. En realidad somos más estrictos que eso, no solo queremos garantizarles los recursos, sino también que su estación de trabajo mantenga los niveles adecuados de capacidad de respuesta a los eventos generados.

La Figura 2.2 muestra el flujo de las tareas de diferentes tipos a los cores del procesador, al igual que en la Figura 2.1, los tipos de aplicaciones están

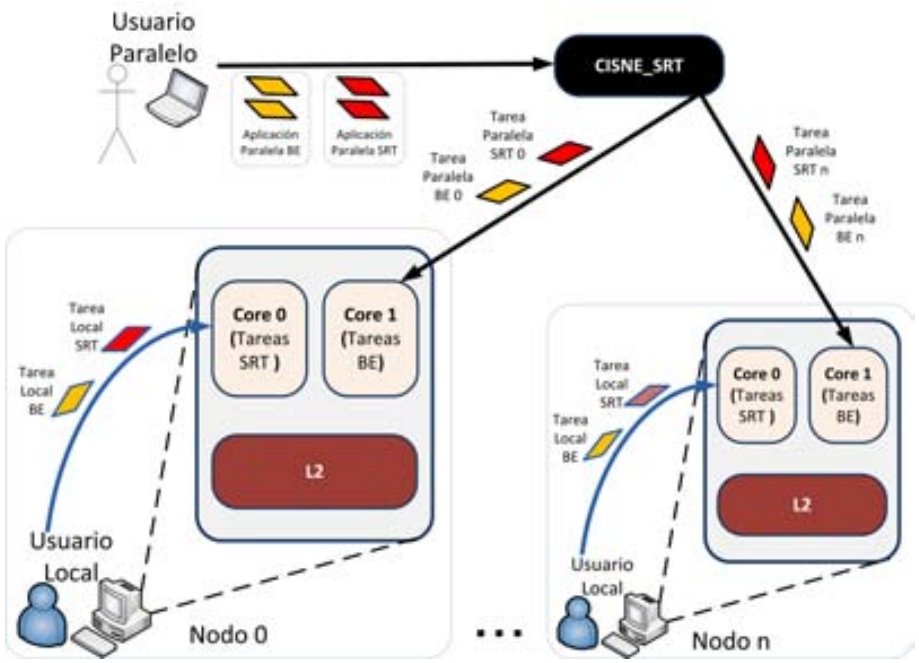

Figura 2.2: Política de asignación de cores por tipo de usuario, denominada *BY_USR*.

representados por colores diferentes, naranja para las Best-effort y rojo para las SRT. Las flechas azules representan los lugares de destino de las tareas de los usuarios locales y las negras el movimiento de las aplicaciones paralelas y sus tareas en el cluster. Como resultado de la aplicación de esta política cada core del procesador tiene solamente tareas pertenecientes a un tipo de usuario.

Como ya hemos mencionado, con esta política intentamos reforzar el Contrato Social, pues le creamos al usuario local una fuente del recurso CPU que solo es accesible a él. Evidentemente, no existe un valor de L_{CPU} determinado para la frontera entre el uso de CPU del usuario paralelo y el local, el usuario local dispone de toda la capacidad de cómputo disponible en el core que le asignemos. Aunque esta política tiene la mencionada ventaja de crear una frontera natural del recurso más preciado, la CPU tiene la desventaja de que hace más compleja la planificación temporal en los nodos del cluster, al mezclar los diferentes tipos de aplicaciones que tomamos en cuenta en este trabajo. Las diferencias entre aplicar la política *BY_USR* y dejar trabajar al planificador de Linux son similares a la de la política *BY_APP*.

2.4.3. Contrato Social por políticas de asignación de cores

En esta subsección del trabajo se resumen los valores empleados en el Contrato Social para las diferentes políticas. La Tabla 2.3 muestra como dividimos los recursos para nuestro caso particular. Cabe destacar que esta división no es genérica, depende tanto de la caracterización de la carga local y paralela como de las características del hardware de los nodos del cluster en uso. Para nuestro caso también tenemos en cuenta que el cluster es homogéneo.

	Contrato Social	
Política	**CPU**	**Memoria Principal**
LIN++	0.85	0.85
BY_APP	0.85	0.85
BY_USR	no	0.85

Tabla 2.3: Límites del Contrato Social empleados, por políticas.

La explicación a los valores mostrados es sencilla, el Contrato Social se aplica siempre en el caso de la memoria principal, con un límite de 0.85, que significa que el 15 % de la memoria principal es accesible para las aplicaciones locales y un 85 % para las aplicaciones paralelas. En el caso del recurso solo aplicamos el Contrato Social a las tareas Best-effort cuando comparten la misma CPU, por lo que para la política BY_USR no está activado. En cambio para la política BY_APP, aplicamos el Contrato Social en el core dedicado a las aplicaciones Best-effort. Es válido destacar que en la política BY_APP las aplicaciones locales SRT reciben toda la CPU que necesitan, pues están asignadas a un core dedicado a el tipo de aplicación SRT.

Capítulo 3

Adaptación del Simulador a las Nuevas Propuestas

En este capítulo describimos nuestro método de simulación, basado en comunicar dos herramientas de simulación diferentes, CISNE y Simulador_Cluster_SRT. También describimos las características más importantes de CISNE, el entorno de planificación que hemos modificado para implementar nuestras propuestas.

3.1. CISNE

CISNE es un entorno de planificación desarrollado en nuestro grupo para la administración de recursos en clusters no dedicados. El entorno tiene como objetivo la ejecución de aplicaciones paralelas en un cluster de ordenadores no dedicados.

El entorno se construye a partir de dos subsistemas, a un nivel más alto encontramos un sistema de planificación espacial, denominado LoRaS, que permite distribuir la carga paralela en el cluster, maximizando la utilización de recursos y la eficiencia de las aplicaciones paralelas. A un nivel más bajo encontramos un entorno de planificación temporal, denominado CSC, encargado de lograr la coplanificación y balanceo de recursos asignados a las tareas pertenecientes a un mismo trabajo paralelo, además de preservar el rendimiento del usuario local. Otra de la características importantes de CISNE es que permite un grado de multiprogramación (MPL) de las aplicaciones paralelas mayor que uno, aún en nodos donde se tiene carga local, sin que esto repercuta negativamente en la capacidad de respuesta de estas últimas.

El entorno CISNE se ha preparado para funcionar en modos diferentes, los cuales son:

- *Desarrollo*: que indica que el sistema se encuentra en modo de evaluación y por tanto lee la carga de aplicaciones a ejecutar de un fichero de configuración.
- *Simulador*: en este caso el sistema no ejecuta ninguna aplicación particular, sino que simula una política de planificación específica para una carga paralela determinada (simulación *fuera de línea*).

Para realizar nuestra experimentación simulada configuramos nuestro entorno para que funcione en el modo *Simulador*, en este modo solo se emplea uno de los dos subsistemas, LoRaS.

3.1.1. Subsistema LoRaS

LoRaS es uno de los dos componentes fundamentales de CISNE, siendo responsable de aceptar peticiones de ejecución de aplicaciones paralelas y definir el mejor momento y lugar para ejecutar tales aplicaciones. La arquitectura del sistema LoRaS es centralizada con demonios en cada nodo para controlar sus estados. Cuando el entorno se lanza en modo Simulador los demonios de control de LoRaS no se lanzan.

La arquitectura del sistema de simulación se divide en dos partes: por un lado la simulación de las políticas de planificación configurables en el entorno, y por otro lado la estimación del tiempo de ejecución de las aplicaciones, una vez que éstas han sido planificadas por el simulador. Cabe destacar que las modificaciones realizadas a LoRaS son las siguientes:

- Soporte para los nuevos tipos de aplicaciones locales y paralelas de tipo SRT.
- Comunicación con el nuevo núcleo de estimación acoplado al sistema (S.K en la Figura 3.1).
- Soporte para procesadores multi-core, a través de la creación e implementación de métodos analíticos con capacidad multi-core.

Para su funcionamiento los núcleos de estimación requieren de dos tipos de información: una caracterización de las aplicaciones a ejecutar (recursos de memoria y CPU consumidos, cantidad de nodos necesarios, tiempo de ejecución en dedicado, etc.) y el estado actual del entorno (cantidad de recursos totales y ocupados en los nodos, cuantas aplicaciones paralelas y en que nodos se encuentran en ejecución, la actividad local en el entorno, etc.). Esta información es provista por los módulos de Caracterización de Aplicaciones y el Gestor de Colas, respectivamente. La utilización del módulo de Caracterización de Aplicaciones permite desligar el proceso de estimación, de valores de caracterización provistos por el usuario paralelo, que suelen encontrarse alejados de la realidad según estudios como [114, 80].

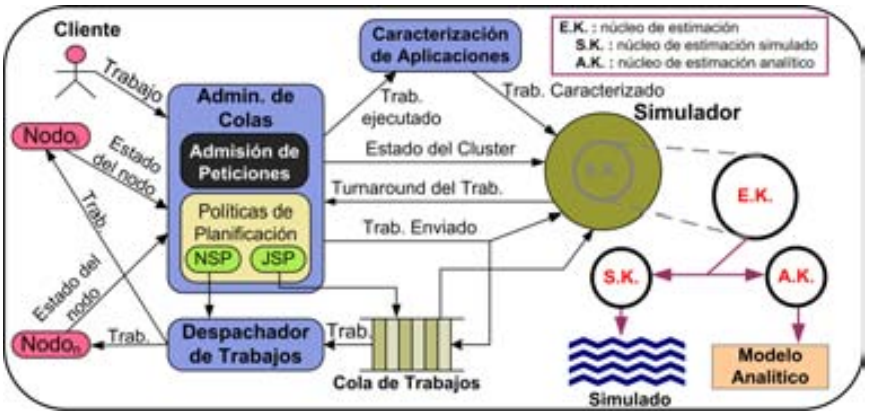

Figura 3.1: Arquitectura del sistema de predicción por simulación integrado en LoRaS.

3.1.2. Arquitectura del simulador fuera de línea

La arquitectura del simulador off-line, se basa en la "duplicación" de módulos, reemplazándolos por otros (dummies), para proveer al gestor de colas de un entorno en el que pueda planificar aplicaciones como si realmente éstas fuesen ejecutadas. Ha de destacarse que todos los módulos dummies se encuentran bajo el control del simulador.

Para realizar las simulaciones off-line en CISNE es necesario proveer en la configuración del sistema información sobre las características del entorno (configuración del entorno en la Figura 3.2). Por ejemplo hemos de entrar en los ficheros de configuración la memoria total, memoria inicialmente en uso, la potencia de la CPU y si existe, cuál es la carga inicial de CPU para cada nodo incluido en el sistema a simular. Esta caracterización es importante, porque en el sistema en producción las características de los nodos son obtenidas por el sistema desde los nodos reales cuando se inicia.

En este caso sin embargo, si tratamos con una simulación el sistema no podrá interrogar a los nodos reales para obtener información sobre sus características y carga actual. De la misma forma y por las mismas razones, también ha de proveerse para la simulación una caracterización de las tareas locales (nodo en que se ejecutarán, tiempo de inicio y fin y consumo de recursos de memoria y CPU) que se tendrán en el entorno a lo largo de la ejecución de la simulación (configuración de la carga Paralela (Local en la Figura 3.2). Finalmente y debido a que el sistema ahora no presta un servicio de planificación a usuarios reales, hemos de proveer la carga de aplicaciones paralelas a ejecutar en ficheros de entrada, como hemos hecho antes con al carga local. Para esto, se provee al sistema de una lista de aplicaciones a ejecutar, junto con el tiempo de arribo de cada una.

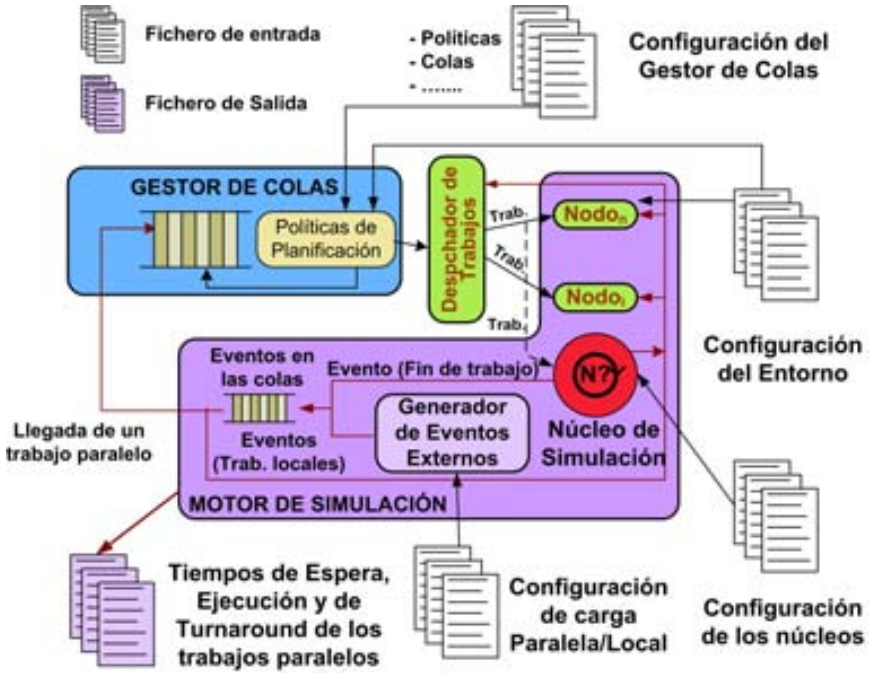

Figura 3.2: Vista modular de la arquitectura del sistema de simulación fuera de línea en LoRaS.

Como se ha descrito en la introducción (sección 1.3.1) de este trabajo, los sistemas RT y SRT necesitan realizar análisis de admisión en el momento de arribo de una tarea SRT, para saber si aceptándola en el sistema no se afectan a las demás tareas. Nuestro entorno, tanto en modo simulado como real, mantiene un estricto control de los recursos CPU y memoria principal en cada nodo. En base a estos datos del nivel de ocupación de recursos de los nodos realizamos nuestro análisis de admisión, que se basa en el control existente de los recursos.

Rechazamos una aplicación paralela SRT si no existe un conjunto de nodos con la cantidad de memoria principal que necesita. En el modo Simulador realizamos este análisis a partir de la información obtenida a partir de los datos obtenidos de los ficheros de configuración en relación a los nodos, presencia y tipo de usuario local y caracterización de las aplicaciones paralelas.

Para el caso del recurso CPU, tomando en cuenta los niveles de aplicaciones SRT que arriban al sistema, que son acotados debido a la naturaleza de las NOWs consideradas en este trabajo, creemos que no es necesario un análisis de admisión exhaustivo. Por esta razón, nos conformamos con realizar un análisis basado en la capacidad de CPU existente en los nodos y la necesaria para la aplicación paralela SRT, para decidir si la aceptamos o no. Cabe destacar que este análisis es complementario al realizado en base a la memoria

principal, por lo que al aceptar una aplicación paralela se tiene en cuenta tanto la disponibilidad de memoria principal como la de CPU.

3.2. Tiempo Remanente de Ejecución

Los dos módulos fundamentales de la arquitectura en modo de simulación off-line son mostradas en la Figura 3.2, el Gestor de Colas (descrito en la sección 3.1.1) y el Motor de Simulación.

El simulador de LoRaS funciona dirigido por eventos discretos, como los son la arribo o finalización de una tarea de cualquier tipo de trabajo. Para realizar las simulaciones, el motor necesita de tres ficheros de configuración:

- **Configuración del Entorno**: En este fichero especificamos el conjunto de nodos a utilizar por el Gestor de Colas para ejecutar la carga paralela. Contiene la cantidad de nodos disponibles y sus principales características (poder de cómputo, tamaño de la memoria principal, cantidad de recursos consumidos por las cargas locales, etc.).
- **Configuración del Núcleo de Simulación**: Permite elegir el método empleado para calcular el valor del Tiempo Remanente de Ejecución (*RExT*, *Remanent Execution Time*) por el Gestor de Colas. Puede ser tanto un método analítico, como el método simulado implementado para este trabajo.
- **Configuración de la Carga**: Contiene la lista de trabajos, tanto paralelos como locales, a simular por el entorno. De cada trabajo se necesita información detallada, que por ejemplo incluye su tiempo de ejecución de caracterización, tiempo de arribo y requerimientos máximos de memoria y CPU.

Una vez que todos los ficheros de configuración se han cargado, el entorno está listo para comenzar el proceso de simulación. Es importante destacar que los valores estáticos utilizados para describir la carga a simular, tales como su tiempo de ejecución de forma aislada o requerimientos máximos de CPU o memoria, son recolectados por el mismo entorno durante las ejecuciones reales para su uso futuro. Toda esta información, junto al estado del cluster, es empleada en generar los conjuntos de datos que conforman las diferentes etapas del proceso de simulación. Hemos de destacar que el evento de llegada de una aplicación paralela marca el comienzo de una nueva etapa, ya que implica una reestimación de los tiempos de turnaround de todas las aplicaciones en ejecución, pues lógicamente todas se verán afectadas por los recursos que esta consumirá. Cada vez que llega una aplicación paralela,

se crea un conjunto de datos que incluye la nueva aplicación paralela y se procesa por el motor de simulación.

Todos los núcleos de estimación que se integran al entorno han de ser capaces de retornar una estimación del Tiempo Remanente de Ejecución ($RExT$, *Remanent Execution Time*) de las aplicaciones paralelas incluidas en el sistema, que es la base del funcionamiento del modelo de estimación del simulador. La idea es la de estimar para un entorno dado, y con un conjunto de aplicaciones en ejecución, cuál es la próxima aplicación que se espera finalice y cuales serán los recursos que se liberarán en tal caso. Los nuevos núcleos de estimación, con capacidad de procesar carga con características SRT, conforman el resto de este capítulo.

El Algoritmo 1 muestra el proceso de simulación de forma simplificada. El primer paso consiste en duplicar el estado del sistema, a los módulos *dummies* creados a tal efecto e inicializar las colas de trabajos y lista de nodos. El control de finalización del algoritmo es la condición del **while** de la línea 3, que controla los trabajos en ejecución. Dentro de este lazo principal se calcula $RExT$ cada trabajo en ejecución (línea 4) y se asume que la próxima aplicación en terminar será J_i, en el instante de tiempo t_i (línea 5). El siguiente paso consiste en actualizar el tiempo de finalización de la aplicación J_i y eliminarla de la cola de aplicaciones en ejecución DRQ. También han de actualizarse los tiempos que las restantes aplicaciones han estado en la cola DRQ. El lazo que se ejecuta entre las líneas 8 y 13 es el encargado de seleccionar las aplicaciones en la cola de espera (DQ) de acuerdo a las condiciones del sistema y las políticas en uso y pasarlas a la cola DRQ.

El núcleo del algoritmo 1 se encuentra en el cálculo del TRE, por tanto es necesario definir algún método para poder realizar la estimación de tiempo remanente de ejecución de un trabajo determinado. A tal efecto se proponen a continuación un conjunto de métodos tanto basados en sistemas analíticos, como en simulación, que nos permiten llevar a cabo tal estimación.

3.3. Núcleos de simulación

En el diseño del LoRaS como simulador off-line, los núcleos de simulación tienen un papel vital, pues son los encargados de proveer los datos para dos puntos críticos del Algoritmo 1, estos puntos son el cálculo de el *tiempo de CPU usado* (Algoritmo 1 línea:7) y el cálculo del $RExT$ (Algoritmo 1 línea:4). Estos datos son importantes porque permiten determinar los progresos que han hecho las aplicaciones paralelas que ya se encuentran en ejecución en la NOW (Algoritmo 1 línea:7) y saber cuál será la próxima aplicación paralela en terminar (Algoritmo 1 línea:4), respectivamente. Cualquier núcleo de estimación que pretendamos incluir en el sistema ha de ser capaz de calcular estos valores.

Algoritmo 1 Proceso de Simulación

1: Duplicar el estado del sistema a *dummy*: siendo DQ una copia de la cola de espera de trabajos, DRQ una copia de la cola de trabajos en ejecución y CL_{sim} una copia de los nodos que conforman el cluster y sus respectivos estados
2: Guardar el momento actual (t_0), como el momento en que la simulación ha comenzado.
3: **while** ($\exists J_k \in DRQ$) **do**
4: **forall** ($J_k \in DRQ$) **do** Calcular $RExT$ de J_k.
5: Asumir que la aplicación J_i es la próxima que finalizará en el tiempo t_i.
6: Actualizar el tiempo de finalización de J_i a t_i (i.e.: calcular el tiempo de ejecución para J_i), y eliminarlo de DRQ.
7: **forall** ($J_k \in DRQ$) **do** Obtener el *tiempo de CPU usado* para $J_k \in t_i - t_0$ y actualizarlo en las respectivas aplicaciones J_k.
8: **while** ($\exists$ recursos disponibles Cl_{sim} y algún trabajo en DQ) **do**
9: Buscar una aplicación $J_x \in DQ$ que pueda ser ejecutada en el estado actual del sistema (Cl_{sim}).
10: Seleccionar el mejor subconjunto de Cl_{sim} para ejecutar J_x, empleando la política del sistema.
11: Ejecutar la aplicación J_x en el subconjunto seleccionado de Cl_{sim} y adicionarla a DRQ.
12: Incrementar el tiempo de espera estimado de J_x en $t_i - t_0$.
13: **end while**
14: **forall** ($J_j \in DQ$) **do** Incrementar el tiempo de espera estimado de J_j en $t_i - t_0$.
15: Asignar $t_0 = t_i$.
16: **end while**

A continuación describimos los enfoques desarrollados en este trabajo para la obtención de los valores necesarios para el funcionamiento de nuestro simulador.

3.3.1. Núcleos analíticos

Para facilitar la comprensión del método analítico propuesto en este trabajo, explicaremos antes otro método similar sin capacidad SRT, que llamaremos CPU. Introducimos también la notación $RExT_{ANL-SRT}$, que denotará nuestro método analítico capaz de realizar estimaciones con cargas SRT.

Nuestro método analítico sin capacidad SRT (CPU) comienza por calcular el $RExT$ que la aplicación necesitaría si se ejecutara de forma aislada, a este valor lo llamaremos $RExT_{isol}(j)$ y se calcula según la Ecuación 3.1.

$$RExT_{isol}(j) = \frac{t_{total}(j) \times (t_{total\_CPU}(j) - t_{used\_CPU}(j))}{t_{total\_CPU}(j)}, \tag{3.1}$$

En esta ecuación, $t_{total}(j)$ es el tiempo total de ejecución, $t_{total\_CPU}(j)$ el tiempo de CPU de la aplicación ejecutada de forma aislada y $t_{used\_CPU}(j)$ el tiempo de CPU que ha empleado desde su comienzo, todos estos valores asociados a la aplicación paralela j .

Ha de destacarse que la Ecuación 3.1 asume que $RExT_{isol}(j)$ es proporcional al tiempo total de ejecución de forma aislada ($t_{total}(j)$) limitado por el tiempo de CPU que consumirá ($t_{total\_CPU}(j) - t_{used\_CPU}(j)$) y el tiempo total de CPU que necesita la aplicación ($t_{total\_CPU}(j)$).

El próximo paso en el método CPU es considerar el porcentaje de CPU requerido por las tareas. De acuerdo a esto, el valor del $RExT(j)$ se calcula de acuerdo la siguiente ecuación:

$$RExT_{CPU}(j) = RExT_{isol}(j) \times \frac{CPU(j)}{CPU_{fea}(j)}, \tag{3.2}$$

donde $CPU(j)$ es el porcentaje de CPU ($t_{total\_CPU}(j)/t_{total}(j)$) que la aplicación puede utilizar y

$$CPU_{fea}(j) = min(CPU(j), \frac{CPU(j)}{CPU_{max}(j)}) \tag{3.3}$$

es el máximo porcentaje de CPU que esperamos la aplicación j consuma. Finalmente

$$CPU_{max}(j) = max(CPU_{par}(n) + CPU_{loc}(n) \mid n \in N(j)) \tag{3.4}$$

donde $CPU_{loc/par}(n)$ es la suma del uso de CPU de cada tarea local/paralela ejecutándose en el nodo n. Destacamos que estos valores representan los requerimientos máximos de uso de CPU (en porcentaje) entre los nodos donde la aplicación paralela j está en ejecución.

Una vez descrito el método $RExT_{CPU}$, lo usaremos como base para el método analítico capaz de estimar considerando cargas SRT. El siguiente método, llamado $RExT_{ANL\_SRT}$, se basa en considerar cuales de los requerimientos de la tareas, ya sea locales o paralelas, son SRT. Para lograr esto, se redefine la expresión $CPU_{fea}(j)$ de $RExT_{CPU}(j)$ para cada una de las políticas creadas de la siguiente manera:

$$C_{fea}(j) = \begin{cases} C(j) & j \in SRT\_Apps \\ min(C(j), C_{vars}(j)) & j \notin SRT\_Apps \end{cases} \tag{3.5}$$

Hemos creado ecuaciones para el cálculo de este valor ($C_{vars}(j)$) para cada uno de las políticas de asignación de cores y otro para simular el comportamiento del sistema operativo. La Tabla 3.1 muestra los números y los nombres de las ecuaciones que sustituyen a $C_{vars}(j)$ en la Ecuación 3.5 de acuerdo a las diferentes políticas.

$C_{vars}(j)$	
política	**Ecuación**
Linux	$C_L(j)$
BY_USR	$C_U(j)$
BY_APP	$C_A(j)$

Tabla 3.1: Sustitución de $C_{vars}(j)$ en la Ecuación 3.5.

La Ecuación 3.6 es la encargada de modelar el comportamiento del SO para una configuración de procesador multi-core con dos cores. Este comportamiento es definido en [82], dónde se establece que cada core recibe un número de procesos balanceado, y que se ejecuta una rutina para balancear el número de procesos entre los cores de forma periódica.

$$C_L(j) = min\left(\frac{C(j) \times (100 - C_{srt}(n)/k)}{C_{n\_srt}(n)/k}\right) \mid n \in N(j) \tag{3.6}$$

En esta ecuación $C_{srt}(n)$ y $C_{n\_srt}(n)$ son la suma de los requerimientos de CPU de cada tarea SRT y no-SRT, respectivamente, en ejecución en el nodo n. Las tareas pueden ser paralelas o locales. La variable k contiene la cantidad de cores del procesador, que es igual a 2 para este caso. $N(j)$ contiene el conjunto de nodos en los cuales la aplicación paralela j están en ejecución. Debido a que la implementación actual de Linux no es capaz

de discriminar con total fiabilidad los procesos en ejecución por su tipo, las tareas migran entres los cores en base a la longitud de las colas, por lo que suponemos que la rutina de balanceo de cantidad de procesos por core hace que la probabilidad de que un proceso determinado vaya a parar a un core sea igual.

Tomando en cuenta la descripción de las políticas mencionada en el capítulo 2, uno de los posibles esquemas de asignación de tareas a los cores es mostrado en la Tabla 3.2. El esquema de asignación de cores representado corresponde a un cluster no dedicado donde se están lanzado aplicaciones paralelas tipo SRT y Best-effort, y el usuario local solo ejecuta aplicaciones locales de tipo Best-effort.

Teniendo en cuenta que nuestras políticas son estáticas, podemos asumir una serie de simplificaciones para crear los modelos analíticos. Para comprender estos modelos es vital tener en cuenta que solo la CPU asignada a las tareas pertenecientes a aplicaciones paralelas de tipo Best-effort es desconocida, pues asumimos que las tareas de aplicaciones paralelas SRT siempre reciben la CPU que necesitan. Esta simplificación está reflejada en la Ecuación 3.5, cuando afirmamos que si $j \in SRT\_Apps$ recibe $C(j)$, donde $C(j)$ es el tiempo de cómputo que necesita la aplicación paralela. Por lo tanto para los requerimientos de las tareas de las aplicaciones paralelas de tipo Best-effort sólo están afectados por las tareas asignadas al mismo core, de acuerdo a la asignación de tareas por core mostradas en la Tabla 3.2.

core/política	***BY_APP***	***BY_USR***
0	par_{SRT}	par_{SRT} & par_{BE}
1	par_{BE} & loc_{BE}	loc_{BE}

Tabla 3.2: Asignación hipotética de las tareas por tipo y core.

Recordamos que la política ***BY_USR*** representa la situación en que cada core del procesador dual-core es empleado para ejecutar las aplicaciones pertenecientes a cada tipo de usuario, ya sea local o paralelo. En esta situación, y tomando como ejemplo la asignación de cores en la Tabla 3.2, las aplicaciones paralelas de tipo Best-effort se verán afectadas por todas las demás aplicaciones paralelas con las que comparte nodo, siendo el nodo más sobrecargado el que ralentiza su ejecución. Para representar esta situación empleamos la Ecuación 3.7 como forma de calcular el valor de $C_{vars}(j)$.

$$C_U(j) = min\left(\frac{C(j)}{C_{p\_n\_srt}(n)} \times (100 - C_{p\_srt}(n))\right) \mid n \in N(j) \qquad (3.7)$$

dónde $C_{p\_srt}(n)$ y $C_{p\_n\_srt}(n)$ son la suma de los requerimientos de CPU de cada proceso *SRT* y *no-SRT* de las aplicaciones paralelas, respectivamen-

te, en ejecución en el nodo n. El razonamiento detrás de esta formulación es el siguiente, cada aplicación paralela j de tipo *no-SRT* es afectada por la carga paralela de tipo *SRT* de su core, razón por la que solo contamos con parte del poder de cómputo de la CPU, $(100 - C_{p\_srt}(n))$. Además nuestra aplicación paralela Best-effort tendrá que competir con el resto de las aplicaciones *no-SRT* (Best-effort como ella) con las que convive en el nodo, para calcular de que porcentaje del poder de cómputo disponible en el nodo dispone empleamos el valor $\frac{C(j)}{C_{p\_n\_srt}(n)}$. Finalmente la capacidad de CPU disponible ha de ser calculada para el peor caso posible, el nodo dónde dispone de menos CPU.

Para la política ***BY_APP***, en la que las aplicaciones se escogen de acuerdo a su tipo a la hora de asignarlas a un core, empleamos la ecuación 3.8 para sustituir el valor de $C_{vars}(j)$ en la Ecuación 3.5.

$$C_A(j) = (100 - max\,(C_p(n) + C_l(n))) \mid n \in N(j) \tag{3.8}$$

dónde $C_p(n)$ es la suma del uso de CPU para cada tarea paralela de tipo Best-effort en ejecución en el nodo n y $C_l(n)$ es la suma del uso de CPU para cada tarea local del tipo Best-effort (no-SRT) en ejecución en el nodo n. En este caso el razonamiento es que si las tareas se asignan por su tipo (SRT o Best-effort), la aplicación paralela de tipo Best-effort se verá afectada por los requerimientos de CPU de las otras aplicaciones paralelas o locales del tipo Best-effort presentes en el nodo n.

Resultados de los métodos analíticos

La Figura 3.3 muestra los resultados para un caso relativamente sencillo, que involucra tres aplicaciones paralelas que arriban al sistema al mismo tiempo ($t = 0$), una de las cuales es de tipo SRT. La métrica empleada es el tiempo de ejecución medido en segundos. La figura contiene los resultados de dos tipos de experimentos, al lado derecho las aplicaciones se lanzaron considerando que en los nodos habían aplicaciones locales de tipo Best-effort en ejecución, y en el lado izquierdo sin presencia de usuario local. Estos resultados muestran claramente la poca fiabilidad del método analítico, con errores promedio que varían entre el 40 y el 60 % .

También es notable la poca diferenciación entre las estimaciones para las diferentes políticas y Linux. Creemos que esta situación se debe a las pautas de diseño de LoRaS, cuyo finalidad es el estudio de diferentes políticas de planificación espacial, por lo que es imposible simular eventos relacionados con la comunicación de las aplicaciones paralelas o la planificación temporal de las tareas en un nodo específico. Para sobreponernos a esta limitación de LoRaS decidimos crear un simulador capaz de tomar en cuenta el comportamiento

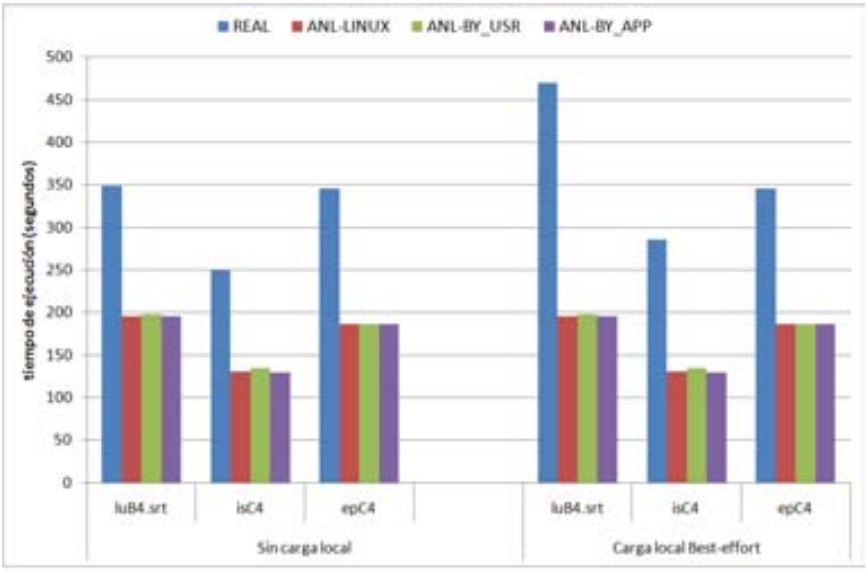

Figura 3.3: Comparación entre el núcleo de estimación analítico (ST-ANL) y las ejecuciones reales (REAL). La carga local de tipo Best-effort está presente en todos los nodos.

individual de las tareas de las aplicaciones para cada nodo del cluster. Con el fin de reusar las potencialidades de LoRaS optamos por perfeccionar la interfaz que brinda para los núcleos analíticos y comunicar los dos simuladores. De esta forma logramos una herramienta de simulación capaz de estudiar tanto políticas de planificación espacial como temporales, que considera aplicaciones SRT en NOWs multi-core. La próxima sección describe el simulador, denominado Simulador_Cluster_SRT, que hemos adicionado al esquema de simulación de LoRaS en lo que hemos llamado E.K. en la Figura 3.1, y que constituye nuestro núcleo de estimación simulado.

3.4. Simulador_Cluster_SRT

Nuestra alternativa al modelo de analítico antes propuesto la constituye un núcleo de estimación basado en la simulación, capaz de procesar tanto carga con características SRT como que los nodos del cluster tengan procesadores multi-core. Como uno de sus objetivos fundamentales es estimar el $RExT$ de las aplicaciones paralelas, lo denotaremos $RExT_{SIM\_SRT}$ y a partir de este punto nos referiremos a este núcleo indistintamente de esta manera o como Simulador_Cluster_SRT.

El Simulador_Cluster_SRT es un programa externo al entorno CISNE, por tanto ha sido necesario establecer una interfaz de comunicación entre los dos

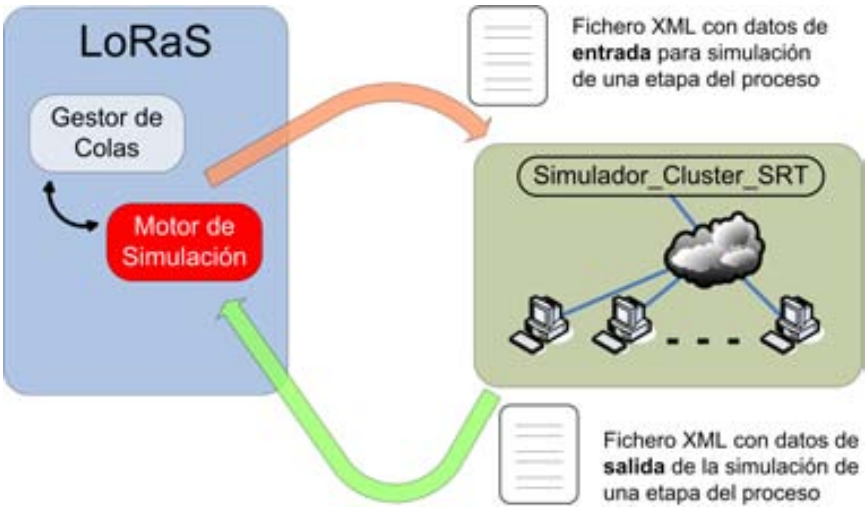

Figura 3.4: Esquema de la simulación a dos niveles.

programas (simuladores), para que el simulador externo pueda recibir los datos de entrada y devolver los resultados. Debido a que los dos programas estaban escritos en diferentes lenguajes de programación, C++ y Java; se optó por comunicarlos mediante ficheros, una vía cómoda y simple.

En nuestro caso, empleamos el formato XML ya que permite crear plantillas y comprobar la consistencia de los datos con facilidad. El funcionamiento de la interfaz mediante ficheros XML es mostrado en la Figura 3.4. La comunicación entre LoRaS en modo de simulación off-line y Simulador_Cluster_SRT ocurre a través de ficheros en formato XML. Como ya se ha explicado anteriormente, la simulación funciona a dos niveles, en el superior, LoRaS genera ficheros XML con el estado del cluster y realiza la ejecución de Simulador_Cluster_SRT. Las razones por las que elegimos XML como formato para los ficheros de datos son:

- La existencia de APIs que facilitan su uso en los lenguajes de programación implicados en el desarrollo.
- Permite comprobar la validez y consistencia de los ficheros de datos de forma rápida y segura. Para cada fichero XML podemos establecer su fichero de formato, contra el cual podemos validarlo y comprobar su consistencia, de acuerdo a la forma en la que lo definimos.

Una vez leídos de los ficheros XML, los datos generados por LoRaS son guardados en clases para posteriormente ser empleados para generar los eventos de arribo de tareas. Durante la duración de la simulación, se guardan los datos que necesita LoRaS en modo de simulación off-line para continuar,

relacionados con el progreso y los tiempos de ejecución remanentes de las aplicaciones paralelas. Estos datos son guardados otro fichero, también en formato XML, para su uso por LoRaS.

Al igual que el método analítico antes descrito, se toma una instantánea del estado del sistema (conformada por el estado del cluster y los datos de las aplicaciones) y se realiza una simulación. Con la diferencia de que el nivel de detalle alcanzado es mucho mayor que en el método analítico. Esto se debe a que este método de estimación del $RExT$ es en realidad un motor de simulación completo; siendo capaz, por ejemplo, de planificar las tareas con diferentes políticas de asignación teniendo en cuenta si las tareas son SRT o no. Cabe destacar que las políticas descritas en el capítulo 2 también pueden ser seleccionadas, además de una implementación que simula el comportamiento de Linux, creada para establecer un punto de comparación.

Hemos de destacar también que $RExT_{SIM\_SRT}$ lleva a cabo su estimación del $RExT$ realizando una simulación del estado de cada nodo, teniendo en cuenta los siguientes factores:

- Los algoritmos de asignación de tiempo de CPU: Estos algoritmos son diferentes si un trabajo es SRT o no, y se eligen de acuerdo al tipo de trabajo y la parametrización de Simulador_Cluster_SRT entre las siguientes opciones:
  - Si el trabajo es SRT, empleamos el algoritmo RMS, explicado en la sección 1.3.2.
  - Si el trabajo es de tipo Best-effort se elige un algoritmo de planificación que emula a Linux, y los trabajos corren durante los intervalos permitidos de acuerdo al tipo de política de asignación de cores y al Contrato Social.
- Los recursos disponibles, calculados en base a las caracterizaciones de las aplicaciones y el estado del cluster.
- Los mensajes intercambiados por las aplicaciones paralelas, explicado en más detalle en la subsección 3.4.3.
- Las políticas de asignación de cores, BY_APP o BY_USR, que están descritas en detalle en el capítulo 2.

En la siguiente sección analizamos un conjunto de casos de uso representativos de aplicaciones SRT contempladas en este estudio y posteriormente describimos la manera en la se administran los principales recursos en nuestro simulador.

3.4.1. Análisis de casos representativos de aplicaciones SRT

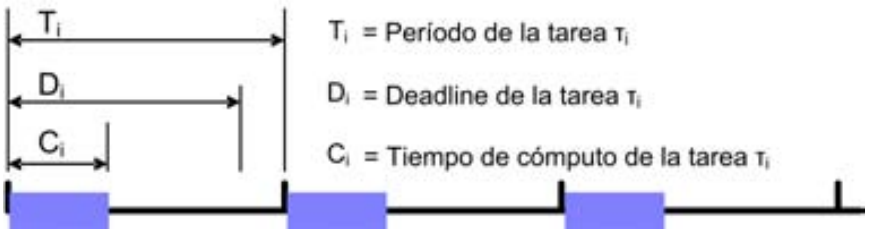

Figura 3.5: Terminología de tareas RT.

La Figura 3.5 muestra la terminología para tareas RT e ilustra el comportamiento del consumo de CPU en el tiempo para este tipo de tareas. En cambio la Figura 3.6 es un ejemplo de aplicación local periódica SRT, en el cual podemos observar que los tiempos de cómputo (los etiquetados como C_i *real*) de la aplicación local varían para cada arribo de un trabajo, aunque mantienen una periodicidad T_i. Este comportamiento se corresponde con los requerimientos de CPU de un vídeo en formato *mpeg*, el cual tiene frames de diferente tamaño, por lo cual las exigencias de C_i son diferentes. Para lograr una correcta predicción y planificación de las tareas Best-effort y SRT tanto locales como paralelas, sería mucho más ventajoso otorgarle la CPU en tiempos conocidos, representados por los C_i *deseado* en la figura.

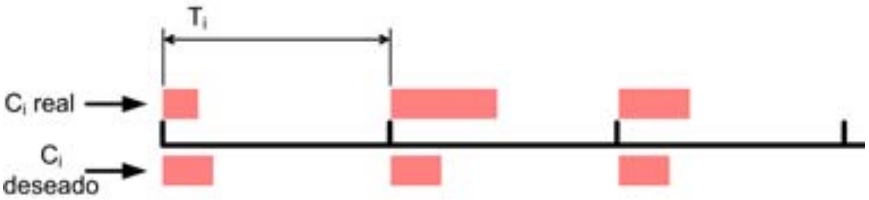

Figura 3.6: Tarea local SRT.

La Figura 3.7 muestra una aplicación aperiódica, que se caracteriza por un tiempo de arribo T_a y un tiempo de ejecución T_{ejec}. Destacamos que consideramos las tareas pertenecientes a una aplicación paralela con turnaround acotado como tareas aperiódicas. En nuestro enfoque otorgamos la CPU de acuerdo a lo representado como C_i *deseado* en la Figura 3.7, y calculamos el valor de C_i de acuerdo a la Ecuación 3.9. Esta ecuación considera el tiempo de ejecución T_{ejec} y un lapso de tiempo como período T_i para obtener el tiempo de cómputo adecuado C_i. Dado que la aplicación paralela con tiempo de turnaround acotado es una aplicación aperiódica, no se ve afectada por este tratamiento. De esta forma hacemos más factible la predicción en nuestro entorno, debido a que este enfoque se emplea tanto en simulación como en las ejecuciones reales.

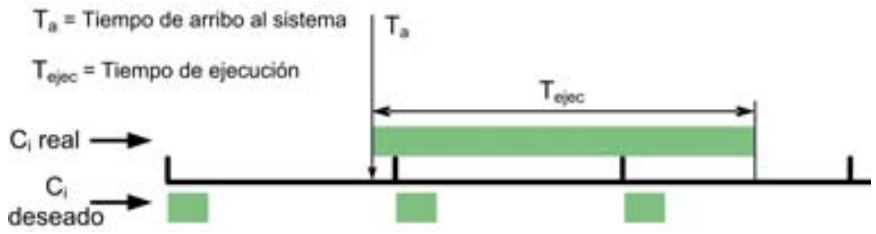

Figura 3.7: Tarea paralela SRT.

3.4.2. Planificación de la CPU

La gestión eficiente de la CPU, uno de los recursos más estudiados tanto en la de la de la planificación de aplicaciones RT como SRT, con la complejidad añadida de que ahora los procesadores son multi-core, merece un apartado en este trabajo. En $RExT_{SIM\_SRT}$ la planificación de la CPU se hace de acuerdo al tipo de tareas y a la política de asignación de cores seleccionada. En el caso de las tareas SRT se emplea el algoritmo de planificación tiempo real RMS; y para los trabajos de tipo Best-effort se planifican de acuerdo a diferentes políticas y criterios de prestaciones. La Tabla 3.3 resume el uso de los diferentes algoritmos de planificación, de tiempo real o no en nuestro simulador, mostrando además si es necesario aplicar el Contrato Social o no.

Política	Alg. RMS	Contrato Social
LIN++	no	si
BY_APP	si, core dedicado a SRT	si, core dedicado a Best-effort
BY_USR	ambos cores	no

Tabla 3.3: Resume de uso del algoritmo de planificación RMS y del Contrato Social por políticas.

3.4.2.1. Admisión de peticiones

La decisión de aceptar o no una tarea en un sistema tiempo real suele recaer en la capacidad de cómputo, en nuestro entorno optamos por tomar en cuenta otro recurso importante, la memoria principal. La ubicación del módulo de Admisión de Peticiones en nuestro simulador es mostrara en la Figura 3.8, LoRaS mantiene un control estricto del uso de memoria en los nodos, tanto en modo simulado como real. Este control nos permite realizar un control previo a la admisión de una aplicación paralela, para detectar si su aceptación provoca que se sobrepase la cantidad de memoria destinada a las aplicaciones paralelas. En caso de sobrepasarse, la aplicación se queda en la cola del cluster, esperando a que se libere suficiente memoria en la cantidad de nodos

que necesita. El control de memoria realizado por el Contrato Social se realiza siempre. Nuestro entorno controla además que la capacidad de cómputo de los nodos no se vea superada, en base al registro que mantiene del uso de CPU.

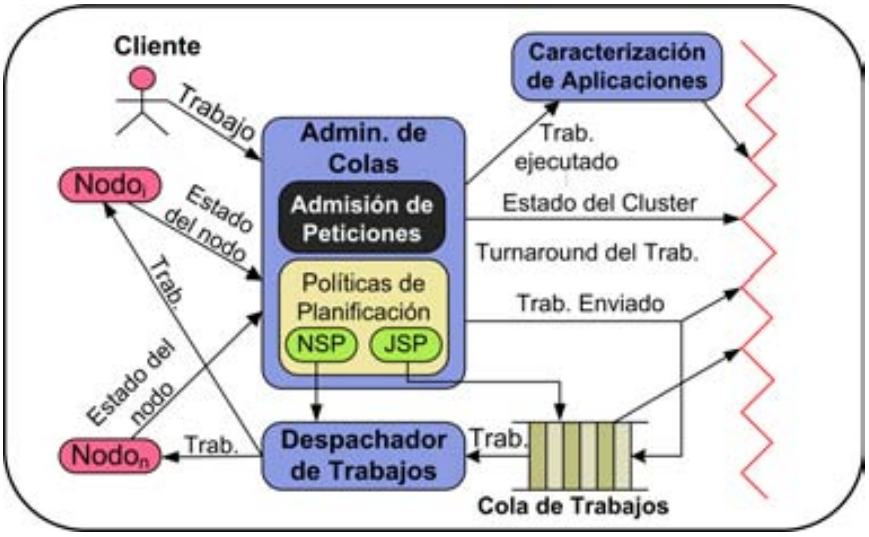

Figura 3.8: Ubicación del sistema de Admisión de Peticiones en LoRaS.

3.4.2.2. Planificación de tareas

La planificación de las tareas se realiza de acuerdo a su tipo, empleándose diferentes formas planificación para diferentes tipos, en el caso de las tareas SRT empleamos el algoritmo RT RMS, y para las Best-effort la política Round Robin. De hecho, el tiempo de cómputo asignado a las tareas Best-effort se puede planificar con alguna de las dos políticas: *Round Robin* o *Coscheduling Cooperativo* [45], el coscheduling cooperativo es una variante de coscheduling que involucra eventos relacionados con el Contrato Social. Nuestro enfoque también respeta el Contrato Social cuando la política de asignación de cores es BY_APP y en el caso de ser BY_USR, esta barrera se crea de forma natural, como resultado del diseño de la política en si misma.

De acuerdo con los análisis antes expuestos, la forma de gestionar la CPU en presencia de aplicaciones SRT se hace de acuerdo a si es local o paralela. En caso de ser local, solo necesitamos garantizarle sus requerimientos de CPU de manera periódica. El problema torna a ser más complicado cuando estamos en presencia de una aplicación paralela SRT, que en nuestro estudio está representada por una aplicación paralela con turnaround acotado. En una aproximación inicial, parecería que es suficiente con asignarle de forma perió-

dica la CPU en cada época (según la definición adoptada en sistemas Linux), con un $C_i = C_{par}(j,n)$ calculado con una ecuación similar a la Ecuación 3.9.

$$C_{par}(j,n) = \frac{T_{par}(j,n) \times (t_{used\_CPU}(j)}{(D(j) - t_{exec}(j))} \times 100. \tag{3.9}$$

Con esta ecuación podemos calcular el tiempo de cómputo ($C_{par}(j,n)$) periódico (cada vez que transcurra el tiempo $T_{par}(j,n)$) que necesitaría la aplicación paralela SRT j en el nodo n para terminar dentro del deadline correcto ($D(j)$). El valor de $t_{used\_CPU}(j)$, al igual que la Ecuación 3.1 de la sección 3.3.1, representa el tiempo de CPU que ha empleado desde su comienzo la aplicación paralela y $t_{exec}(j)$ es el tiempo que ha pasado en ejecución la aplicación paralela j desde su comienzo.

Sin embargo, aunque este enfoque permite reservar tiempo de cómputo para una aplicación paralela, es en extremo pesimista. Con un valor calculado en base al deadline deseado, sólo se logra que la aplicación paralela SRT termine cerca del valor usado como base del cálculo. La Figura 3.9 muestra dos casos de asignación del quantum de la CPU. En esta figura C representa el valor de tiempo de cómputo reservado a una aplicación paralela SRT (calculado con la Ecuación 3.9), R el slice del quantum reservado a otras aplicaciones SRT locales o paralelas y S_{libre} el segmento de quantum de CPU que no está en uso por ninguna aplicación SRT. En el caso **A** podemos notar que la aplicación paralela podría recibir un slice mayor, representado por T_{max_disp}. El caso **B** muestra un comportamiento más agresivo, en el que se aprovechan mejor los recursos. Nuestro simulador puede funcionar de ambas formas, tanto para simulación como para las ejecuciones reales, comportamiento que se activa en el momento de su inicio. Cuando se intenta aprovechar al máximo los recursos la aplicación paralela SRT que ha arribado antes recibe el máximo slice del quantum posible, y cuando esta termina la siguiente por orden de llegada pasa a recibir el mismo tratamiento. Empleando este enfoque, las aplicaciones paralelas SRT pueden tomar ventaja de cualquier momento de baja carga en los nodos y finalizar antes.

3.4.3. Gestión de Memoria y Red

Ha de destacarse que aún cuando LoRaS entrega una carga balanceada al núcleo de estimación simulado, teniendo en cuenta el estado del nodo y sus recursos, Simulador_Cluster_SRT es capaz de controlar la memoria usada en el nodo de acuerdo al Contrato Social definido en el momento de su ejecución.

El recurso Red también ha sido tomado en cuenta en el diseño del núcleo simulado. En nuestro simulador, las aplicaciones paralelas generan mensajes de

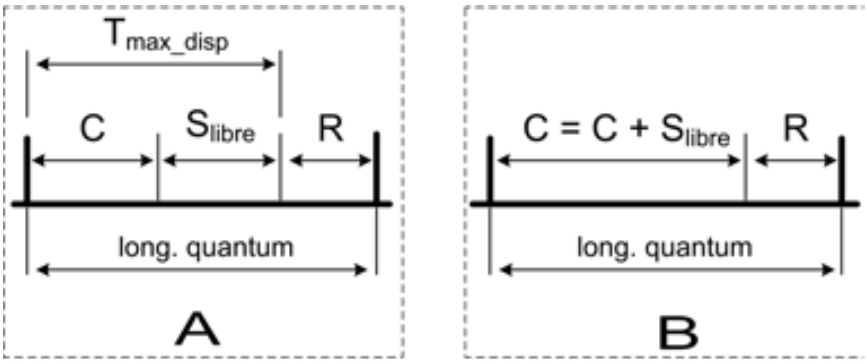

Figura 3.9: Asignación dinámica del quantum de CPU en el núcleo simulado.

acuerdo a su caracterización. Logramos esto empleando una implementación booleana de la distribución de Bernoulli, que inicializamos con el resultado de $100 - getCPUUsage(job)$, basándonos en la consideración de que el tiempo que no se gasta en cómputo se gasta en comunicaciones.

En base al valor generado por la distribución de Bernoulli, podemos decidir si la aplicación comunica o no. En caso de que comunique, generamos mensajes para ella en todos los nodos en los que hay trabajos de la aplicación paralela y los guardamos en el buffer local del nodo que los genera. Posteriormente cuando la aplicación paralela para la cual generamos los mensajes tiene asignada la CPU, revisa el buffer buscando mensajes y los envía a los nodos donde están el resto de los trabajos. Estos mensajes se guardan en los buffers de los nodos remotos, siendo este proceso retrasado para simular la demora de la red. A continuación cuando las tareas en sus respectivos nodos tienen la CPU, procesan los mensajes.

Capítulo 4

Entorno de Planificación

En este capítulo describimos las modificaciones realizadas a CISNE como parte del proceso de actualización para lograr nuestros objetivos, dedicando especial atención al middleware SRT_Scheduler, una capa intermedia creada para mediar entre CISNE y el sistema operativo. Nuestro interés se centra en incluir en el entorno un nuevo nivel de planificación temporal, por encima del sistema operativo, que se ocupe de administrar el reparto de los threads a los cores, la afinidad de los mismos y los aspectos de bajo nivel que tienen que ver con las necesidades de las nuevas aplicaciones SRT, y que no proporcionan los sistemas operativos convencionales. Aspectos como la reserva de recursos, el balanceo de las colas de threads en función de su tipología (IO o CPU bound), el reparto de los recursos CPU y memoria principal en función del tipo específico de tarea SRT (periódica o aperiódica), están en el punto de mira de nuestras aportaciones. Hemos denominado CISNE_SRT a la unión de el sistema CISNE con SRT_Scheduler.

4.1. Arquitectura

El diseño original de CISNE incluye un subsistema denominado CSC [45], cuyos principales objetivos eran implantar el coscheduling cooperativo y controlar el cumplimiento del Contrato Social. El subsistema CSC, necesita acceso a llamadas al sistema para realizar correctamente su cometido. Por esta razón nos hemos visto obligados a sustituir este subsistema por otro, que hemos llamado SRT_Scheduler, al que hemos diseñado para realizar la tarea de la planificación temporal considerando los problemas que han originado este trabajo: la presencia de aplicaciones SRT y los procesadores multi-core. SRT_Scheduler está completamente implementado en espacio de usuario, lo que permite que nuestro sistema pueda ser usado por un usuario sin privilegios en Linux.

Destacamos que algunas de las características más importantes de CSC fueron migradas al subsistema LoRaS, más concretamente al demonio de control de LoRaS, denominado LoRaSd. Las características más importantes migradas fueron el mecanismo de control e interacción de trabajos, y el de control del MPL. Una de las características más notorias de CSC de la que no disponemos en la actualidad es la coplanificación cooperativa, debido a que controla eventos imposibles de detectar sin realizar llamadas al sistema. Otro de los objetivos de CSC, la asignación uniforme de los recursos a lo largo del cluster, es reforzado desde el nodo servidor de LoRaS, en base a la información de estado de los nodos que dispone.

4.1.1. LoRaS

Como hemos mencionado previamente, LoRaS es un componente fundamental de CISNE, responsable de aceptar peticiones de ejecución de aplicaciones paralelas y definir el mejor momento y lugar para ejecutar tales aplicaciones. La arquitectura definida para el sistema CISNE puede observarse en la figura 4.1.

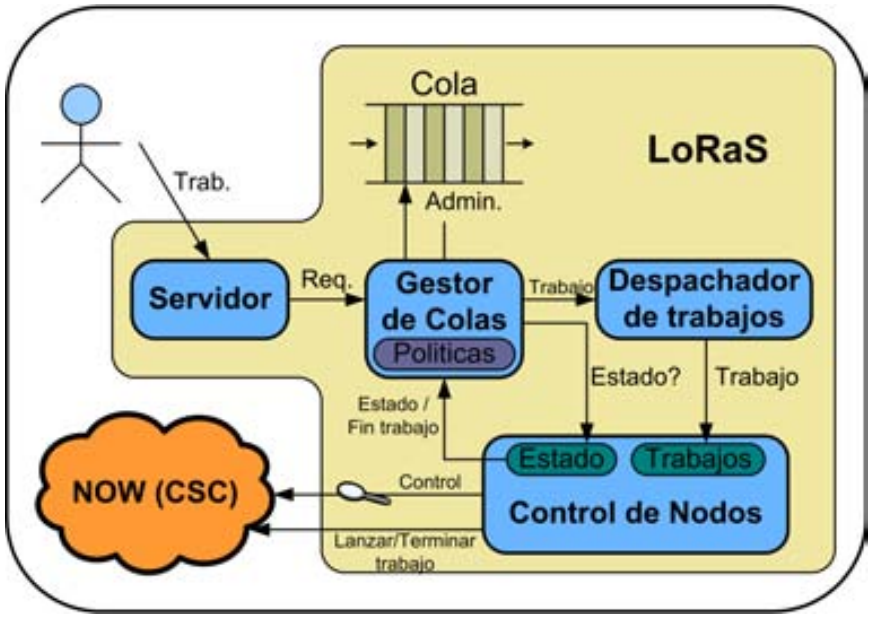

Figura 4.1: Arquitectura del sistema CISNE, interacción entre los subsistemas LoRaS y CSC.

La arquitectura propuesta emplea un control central que es el responsable de admitir y administrar los trabajos que se envían para ser ejecutados. Posteriormente en cada nodo existe un demonio de control, denominado LoRaSd, encargado de controlar las aplicaciones lanzadas en ellos y el estado del nodo

en cuanto a la utilización de recursos, evaluando tanto los empleados por parte de las aplicaciones paralelas, como por parte de los usuarios locales. En la figura 4.1 pueden observarse a nivel funcional, cuáles son los bloques que componen el sistema y cómo interactúan. Las responsabilidades pertenecientes a cada uno de los bloques presentados en la arquitectura de LoRaS son:

- *Servidor*: es el encargado de recibir una petición por parte de un cliente, proveniente de cualquier lugar de la red, para ejecutar un trabajo. Cada vez que esto sucede se debe dar formato al requerimiento para generar un objeto aceptable por parte del gestor de colas, y posteriormente entregado a éste para que pueda ser servido.

- *Gestor de Colas*: es el centro del esquema y es el responsable de recibir los trabajos a servir, y administrar su ejecución en función de las políticas que se hayan definido. Es responsable de la administración de la cola de trabajos y el despacho de trabajos a ejecución. Cabe destacar, que este módulo debe interactuar con el módulo de control de nodos para obtener aquella información de estado que le sea necesaria para tomar sus decisiones de planificación.

  - *Políticas*: este módulo incluido en el gestor de colas es el que implementa todas las políticas utilizadas por el gestor. Entre ellas se tienen: las políticas de *selección de nodos*, junto con la elección de qué nodo del grupo será sobre el que se lance la aplicación en primera instancia; los criterios de *selección de aplicaciones* de entre todas las existentes en la cola, y las políticas mediante las cuales se han de *ordenar los trabajos* en la o las colas administradas.

    Las decisiones de planificación se toman cada vez que se produce un evento de planificación, lo que se da en 3 casos: cuando llega una aplicación al sistema, cuando finaliza una aplicación lanzada por el sistema, o regularmente en base a un período fijo de tiempo.

- *Despachador de trabajos*: se encarga de recibir un trabajo del gestor de colas y posteriormente envía la información necesaria para su ejecución al controlador del nodo donde se ha de lanzar la aplicación.

- *Control de Nodos*: se encuentra implementado como un demonio en cada nodo del cluster y tiene dos responsabilidades claramente definidas: por un lado el control del estado del nodo y por otro el control de las tareas que se han lanzado sobre él.

  - *Estado*: es el responsable de obtener datos del estado del nodo en lo referente a si existe o no actividad local, y si existe, que

porcentaje de recursos se están utilizando. Así mismo debe monitorizar la cantidad de memoria disponible y la utilización total de CPU.

- *Trabajos*: las responsabilidades que le corresponden a este módulo tienen que ver con lanzar, y mantener la pista de las aplicaciones lanzadas en el nodo por parte del sistema.

El sistema se ha preparado para funcionar en modos diferentes:

- *Desarrollo*: que indica que el sistema se encuentra en modo de evaluación y por tanto lee la carga de aplicaciones a ejecutar de un fichero de configuración. En este caso el módulo servidor de la figura 4.1 no es utilizado y por tanto su interfaz de recepción de eventos remotos de usuarios se encuentra desactivada.

- *Simulador*: en este caso el sistema no ejecuta ninguna aplicación particular, sino que simula una política de planificación específica para una carga paralela determinada (simulación *fuera de línea*).

El modo empleado durante las ejecuciones reales es del desarrollo.

4.2. SRT_Scheduler

Nuestra propuesta para la creación del nuevo nivel de planificación temporal, que se ocupe de administrar el reparto de los threads a los cores empleando la afinidad y controlar los aspectos de bajo nivel que tienen que ver con las necesidades de las nuevas aplicaciones SRT se denomina SRT_Scheduler.

4.2.1. Objetivos de SRT_Scheduler

Los objetivos del subsistema SRT_Scheduler son los siguientes:

- Realizar la planificación temporal de los tipos de aplicaciones previamente descritas en este trabajo, que comprende tanto las tareas SRT como las Best-effort. Cabe destacar que de acuerdo a la política de planificación de cores tendremos que planificar solo tareas SRT o una mezcla de tareas.

- Implantar las políticas de asignación de cores definidas en el capítulo 2, y permitir que nuevas políticas sean implementadas con un esfuerzo razonable.

- Administrar las potencialidades brindadas por los procesadores multicore, tales como la asignación de tareas a un core definido.
- Estimular la coplanificación de las aplicaciones paralelas, tomando en cuenta que estamos limitados a trabajar en el espacio de usuario.

Para lograr cumplir estos objetivos hemos creado un sistema con las siguientes características.

4.2.2. Arquitectura

La Figura 4.2 muestra la interacción entre CISNE y SRT_Scheduler, y la posición de intermediario entre el sistema operativo que ocupa este último. Otro punto de interés es el relacionado con la aceptación de un nuevo trabajo paralelo SRT en nuestro entorno de planificación. Dicha decisión es tomada en el nodo que actúa como servidor en nuestro entorno, que es capaz de realizar un Análisis de Viabilidad que considere tanto las necesidades de cómputo de las aplicaciones paralelas SRT como el estado de la memoria principal.

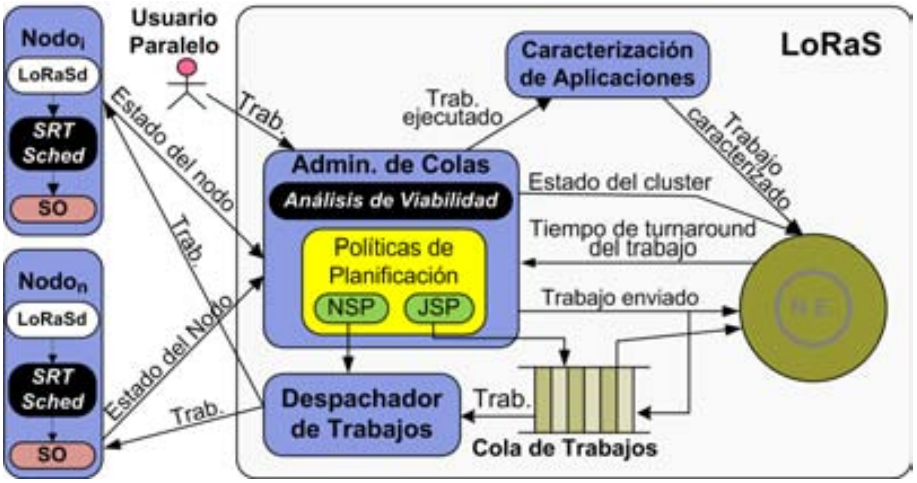

Figura 4.2: Interacción entre CISNE y SRT_Scheduler.

La comunicación entre el subsistema LoRaS y SRT_Scheduler se logra con el demonio de control como intermediario, a través de un sistema de comandos que pueden ser enviados a SRT_Scheduler. Los comandos actuales son los siguientes:

- *ADD_TASK*: encargado de adicionar una tarea, de acuerdo a los datos de la tarea aportados como parte del comando, el planificador es capaz de reconocer el tipo de aplicación y asignarlo a uno de los cores de acuerdo a la política en uso.

- *CHANGE_ TASK_ AFFINITY*: implementado con fines futuros, recibe como parámetros un identificador de proceso y un índice de core, cambia la afinidad de la tarea al índice recibido como parte del comando.
- *SET_ TASK_ PRIO*: cambia la prioridad de la tarea.
- *END_ SERVICE*: detiene el planificador, dejando todas las tareas que se encuentren en ejecución en el momento de recibir el comando con señal SIG_CONT.

A través de este esquema somos capaces de realizar las operaciones relacionadas con la administración del potencial de los procesadores multi-core. Otra ventaja de esta arquitectura es que podemos enviar comandos al planificador desde cualquiera de los dos niveles de CISNE, lo mismo desde el servidor centralizado o desde el demonio de control. El hecho de que el demonio de control pueda administrar también la planificación de las tareas puede facilitar que en un futuro el entorno sea capaz de reaccionar ante situaciones locales a los nodos, tales como que se ejecute alguna tarea local no caracterizadas con un mayor consumo de memoria y sea necesario detener alguna de las aplicaciones paralelas en ejecución.

4.2.3. Planificación temporal

La planificación temporal de los tipos de aplicación Best-effort y SRT, ya sean locales o paralelas, que existen en nuestro escenario, es uno de los objetivos más importantes de SRT_Scheduler. Para realizar esta planificación ha de tener en cuenta la política de asignación de cores en uso, que recibe como parámetro de entrada en el momento de su ejecución.

Para la planificación las tareas SRT empleamos una variante simplificada del algoritmo RMS [72], algoritmo basado en prioridades estáticas. Las razones para decantarnos por este algoritmo las encontramos en el trabajo [14]. En este trabajo proponen una variante del algoritmo RMS que, según los autores del mismo, tiene un comportamiento mejorado para tareas RT con tiempos de cómputo variables, pues utiliza estadísticas para predecir el tiempo de cómputo, en vez de emplear siempre el peor caso o una media de los tiempos de cómputo. Otra de las razones que tenemos es la consideración de que todo nuestro trabajo de planificación ha de ocurrir en el espacio de usuario, esta situación nos obliga a emplear algoritmos con el menor overhead posible, siendo la variante simplificada del RMS que implementamos una buena opción. En nuestro simulador están implementados tanto el algoritmo EDF como el RMS, pero durante la experimentación hemos empleado siempre RMS debido a que en el entorno real solo hemos implementado el RMS.

Para los algoritmos basados en prioridades, las tareas aperiódicas son planificadas como trabajos en background, o puede emplearse un servidor para tareas aperiódicas con una capacidad y prioridades bien definidas. Nuestro enfoque es diferente, pues convertimos las tareas aperiódicas en periódicas empleando la Ecuación 4.1.

$$C_{parSRT}(j) = DEF\_Q \times \frac{J_{Caract}(j)}{D(j)}. \tag{4.1}$$

dónde $J_{Caract}(j)$ son los jiffies consumidos por la aplicación paralela SRT durante su caracterización, $D(j)$ el tiempo acotado de turnaround para la aplicación paralela SRT, convertido a jiffies, y DEF_Q es el quantum por defecto empleado por SRT_Scheduler durante su proceso de planificación.

En nuestro planificador tenemos dos políticas de asignación de cores que implican diferentes formas de planificar las tareas.

- *Política* BY_APP: como ya mencionamos en el capítulo 2, la política BY_APP asigna las tareas a los cores de acuerdo a su tipo, ya sea SRT o Best-effort. Como mencionamos antes, esta política ha sido diseñada para mejorar las prestaciones recibidas por las aplicaciones SRT, por lo que todas las tareas SRT van a un core predefinido y se planifican de acuerdo al algoritmo RMS. Las tareas de tipo Best-effort han de competir libremente por los recursos de cómputo existentes en el otro core, respetando los límites del Contrato Social.

- *Política* BY_USR: en este caso el proceso de planificación es un poco más difícil, pues las tareas SRT coexisten con las de tipo Best-effort en el mismo core. Nuestro planificador funciona de igual forma para ambos tipos de usuario, garantizando las necesidades de cómputo de las aplicaciones SRT y repartiendo el resto de la CPU disponible de forma equitativa entre las tareas de tipo Best-effort que estén asignadas a dicho core. En este caso también empleamos el algoritmo RMS para la planificación de las tareas SRT.

4.2.4. Sistema de prioridades

El sistema de prioridades interno de nuestro planificador se basa en los tipos de las aplicaciones y en los usuarios que las lanzan, si esta información coincide, la prioridad de la aplicación es la misma. La Tabla 4.1 muestra el orden de prioridades de nuestro sistema de prioridades interno, implementado a través de las opciones disponibles a nivel de usuario sin privilegios de Linux. La Figura 4.3 ilustra el uso consumo del quantum para las diferentes aplicaciones y nuestro planificador.

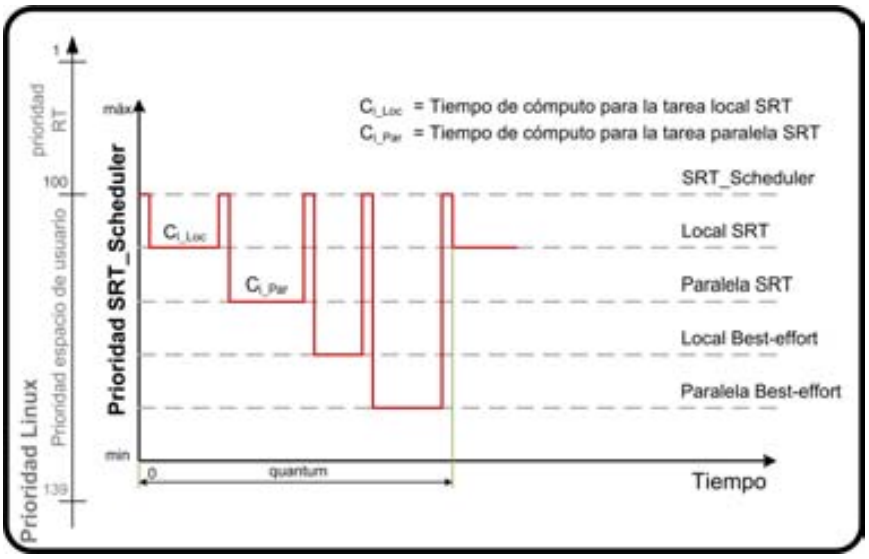

Figura 4.3: Prioridad y reparto del quantum en SRT_Scheduler.

Lógicamente, si esperamos que nuestro planificador sea capaz de controlar el flujo de las aplicaciones, necesita disfrutar de una prioridad mayor que los procesos que planifica. En la Figura 4.3 se puede constatar que nuestro planificador recibe la máxima prioridad disponible en el espacio de usuario, y el resto de las prioridades internas se ajusta a valores correspondientes en las prioridades de usuario de Linux. También en esta figura mostramos un ejemplo de asignación del quanto por nuestro planificador, mostrando como se hace uso del tiempo tomando en cuenta las prioridades internas. Las prioridades que reciben las aplicaciones a planificar están separadas por un intervalo de 3 niveles de prioridad, con el objetivo de paliar parcialmente los problemas que crea el sistema dinámico de asignación de prioridades de Linux a nuestro planificador. El planificador se reestablece su prioridad cada vez que recibe una petición de adicionar una aplicación paralela a sus colas de ejecución.

Sobre este punto cabe destacar que existe la opción de adelantar una de las aplicaciones paralelas SRT, aprovechando el slice del quantum libre en cada ciclo del planificador, como se muestra en la Figura 4.4. En esta figura $C(j)$ es el tiempo de cómputo de la aplicación j a adelantar, $T_{max\_disp}(j)$ el tiempo máximo disponible para la aplicación j, resultado de la suma de su tiempo de cómputo más S_{libre} y R representa el segmento del quanto reservado a otras aplicaciones, ya sea SRT y Best-effort. En el caso de las aplicaciones Best-effort la reserva se hace a través del Contrato Social. La aplicación paralela a adelantar se elige por el orden de llegada al sistema. Como todas las aplicaciones paralelas que llegan al sistema tienen la misma prioridad interna, quedan ordenadas en la cola de planificación por su orden de llegada, siendo

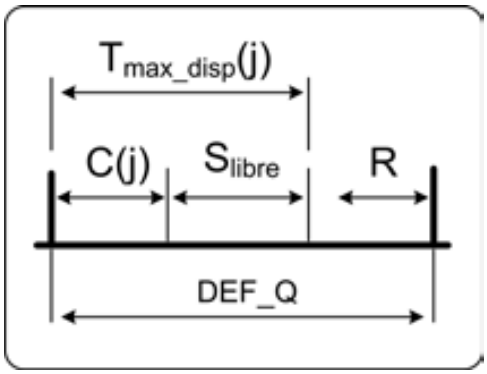

Figura 4.4: Asignación dinámica del quantum de CPU en el núcleo simulado.

posible seleccionar la siguiente de forma fácil y sin sobrecarga adicional al sistema. Finalmente destacar que hemos adelantado aplicaciones paralelas durante la experimentación mostrada en este trabajo con este principio tanto para las ejecuciones reales como las simuladas.

Tipo de Usuario	Tipo de Aplicación	Prioridad
local	SRT	1
paralelo	SRT	2
local	Best-effort	3
paralelo	Best-effort	4

Tabla 4.1: Sistema de prioridades interno del middleware creado

Otro factor a tener en cuenta a la hora de asignar una prioridad interna a las aplicaciones es el relacionado con la longitud del quantum del sistema operativo, cuyo comportamiento describimos en la sección 2.1. Como podemos deducir de lo expuesto, tanto el quantum como la prioridad son dinámicos, dependiendo de los valores por defecto y el comportamiento de las aplicaciones. Sopesando la información disponible sobre el manejo de los quantums de Linux e impulsados por nuestras propias necesidades hemos establecido un valor fijo para el uso interno de nuestro planificador. Este valor se emplea como base en la Ecuación 4.1, empleada para el cálculo del tiempo de cómputo de las aplicaciones paralelas SRT, y de forma interna en nuestro planificador, y tiene un valor de 200 milisegundos.

4.2.5. Coplanificación en SRT_Scheduler

Como ya mencionamos en el capítulo introductorio de este trabajo, existen tres variantes de coplanificación, la explícita, la implícita y la híbrida. La coplanificación explícita es poco apropiada para entornos no dedicados, y la

implícita conlleva la detección y monitorización de eventos solo accesible a través de llamadas al sistema. El no poder realizar coplanificación explícita también nos imposibilita la coplanificación híbrida, ya que es la combinación de la explícita con la implícita.

La alternativa que nos queda es estimular la probabilidad de ocurrencia de la coplanificación, de una forma similar a la *coplanificación predictiva* [97, 94], que es una variante de la coplanificación explícita. En nuestro entorno podemos controlar dos variables relacionadas con la coplanificación, el grado de multiprogramación (MPL) y la prioridad de las aplicaciones paralelas. Veamos como podemos conjugar estos factores para estimular la coplanificación.

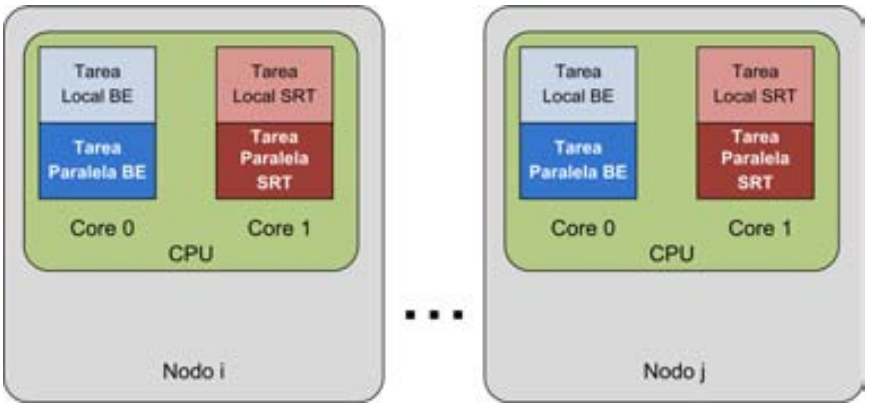

Figura 4.5: Posible distribución de las tareas en los nodos de un laboratorio de universidad.

Como hemos descrito en la sección 4.2.4, las aplicaciones de una misma clase gozan de la misma prioridad y al combinar esta característica de nuestro planificador con ciertas heurísticas implementadas en nuestro sistema, podemos obtener una probabilidad del 50 % de coplanificación de las aplicaciones paralelas. Las características son las siguientes:

- Las colas de aplicaciones paralelas en un entorno como el nuestro, una universidad, no son para nada parecidas a las que podríamos encontrar en entornos productivos, donde pueden llegar a haber cientos de aplicaciones esperando ser lanzadas. Creemos que la situación que más se asemeja a la nuestra es en la cual las colas son cortas, con pocas aplicaciones y tiempos de ejecución largos.

- El comportamiento de los usuarios locales es conocido y similar. Considerando que nuestro cluster no dedicado está conformado por ordenadores aula de laboratorio cerrada, somos capaces de conocer su comportamiento, tanto en cantidad de recursos como en frecuencia de

uso. En estas condiciones todos los alumnos realizarán una actividad similar.

Conjugando estas características con el control del MPL, posible a través de CISNE, cuyo valor máximo para las aplicaciones paralelas estableceremos en 2. Con esta configuración logramos que solo pueda coexistir dos aplicaciones paralelas (ya sean SRT o Best-effort) en cada nodo del cluster, y tomando en cuenta que nuestra planificación espacial está configurada de tal forma que siempre envía las tareas de una misma aplicación paralela a diferentes nodos, logramos que las probabilidades de coplanificación se vean incrementadas.

La Figura 4.5 muestra un ejemplo, para el cual tenemos una probabilidad de coplanificación del 50 %. Esto se debe a que si como máximo podemos encontrar 4 tareas en ejecución en un nodo, dos locales y dos paralelas, las probabilidades de que una de las dos aplicaciones paralelas tenga la CPU es del 50 %. Esta probabilidad se ve favorecida además por el hecho de que las mismas clases de aplicaciones tienen la misma prioridad a lo largo de todo el cluster.

Capítulo 5

Implementación de las Propuestas

Las modificaciones a CISNE para adaptarlo a los nuevos requerimientos han llevado a la creación de dos propuestas, cuyo diseño e implementación detallamos en este capítulo. Las propuestas van dirigidas a permitir que el entorno sea capaz de planificar aplicaciones paralelas SRT, considerando que la NOW sea multi-core, y extender nuestro simulador en la misma forma.

5.1. Simulador_Cluster_SRT

En esta sección describiremos las modificaciones, en forma de extensiones, añadidas al sistema CISNE. Hemos de destacar que las extensiones sólo cubren el modo de simulación off-line (descrito en la sección 3.1.2) del sistema. Este grupo de modificaciones va dirigido a estudiar el comportamiento de las NOWs frente a los nuevos tipos de cargas SRT, considerando que los nodos de la NOW poseen procesadores multicore.

Como ya hemos mencionado antes (Capítulo 3.1), el núcleo de estimación simulado es una aplicación independiente al motor superior de simulación (LoRaS). Esta aplicación recibe el nombre de Simulador_Cluster_SRT y sus características principales son mencionadas en la sección 5.1.1. En este capítulo describiremos su arquitectura e implementación.

5.1.1. Arquitectura

La arquitectura del Simulador_Cluster_SRT está basada en una plataforma orientada a objetos diseñado para los programadores que usan Java y desarrollan modelos de simulación. A partir de esta plataforma se estructura todo el modelo empleado, razón por la cual comenzaremos por introducirlo.

5.1.1.1. Plataforma DESMO-J

Según [64], DESMO-J es una plataforma orientada a objetos diseñando para los programadores que desarrollan modelos de simulación. "DESMO-J" significa "*Discrete-Event Simulation and MOdelling in Java*" (Simulación dirigida por eventos discretos y modelado en Java). Esta forma de nombrar la plataforma destaca las dos características más significativas de DESMO-J:

- DESMO-J funciona bajo el paradigma de la simulación dirigida por eventos discretos. En modelos de este tipo, todos los cambios de estado del sistema se supone sucederán en puntos discretos del tiempo. Entre dichos acontecimientos, el estado del sistema se asume seguirá siendo el mismo. La simulación dirigida por eventos discretos es por lo tanto, particularmente conveniente, para los sistemas en los cuales los cambios del estado relevantes, ocurren de forma repentina e irregular.
- DESMO-J está implementado en Java. Usar esta plataforma para construir modelos de simulación implica la escritura de un programa en Java.

DESMO-J ha sido desarrollado en la Universidad de Hamburgo y en la actualidad es mantenido por un equipo de investigadores [28]. Esta plataforma adiciona características que simplifican el desarrollo de simuladores dirigidos por eventos discretos. Entre ellas podemos mencionar:

- Clases para modelar componentes comunes de los modelos, como por ejemplo: colas y distribuciones estocásticas basadas en números aleatorios.
- Clases abstractas que pueden ser adaptadas a comportamientos específicos (modelos, entidades, eventos, procesos de simulación y otras).
- Una infraestructura de simulación lista para emplearse que comprende los planificadores, lista de eventos y reloj de simulación, todas encapsuladas en una clase llamada *Experiment*.

Cabe destacar que esta última clase denota una separación entre el modelado y la experimentación, lo cual facilita su uso. Todas las clases están contenidas en paquetes de Java para organizarlas y hacerlas más accesibles.

En el Simulador_Cluster_SRT se emplean parte de las clases brindadas por esta plataforma, siendo las principales las que se muestran a continuación:

- *public abstract class* ***Entity***: representa la superclase para todas las entidades en un modelo. Se supone que las entidades serán programadas

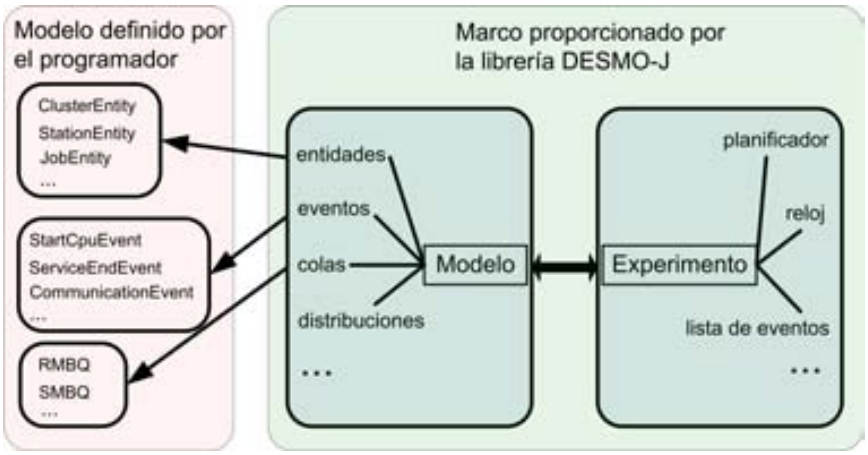

Figura 5.1: Interacción Modelo-Experimento en DESMO-J.

en cierto punto de simulación de acuerdo a eventos compatibles.
Las clases que heredan de Entity encapsulan usualmente toda la información de entidades del modelo relevantes al modelador. Empleando los eventos, podemos cambiar el estado del modelo en cierto momento programable del tiempo.

- *public abstract class* ***Event***: provee la superclase para eventos definidos por el usuario que pueden cambiar el estado del modelo. Al ser una plataforma dirigido por eventos, los cambios de estado son generados por eventos que son programados en distintos puntos del tiempo de simulación. Un evento puede actuar solo en una entidad, cambiando su estado de acuerdo a la reacción programada de la entidad al evento específico.

- *public abstract class* ***Model***: las clases que heredan de esta superclase contienen todas las referencias a todos los componentes del modelo a simular.

Una vez dados a conocer los elementos básicos necesarios para comprender la plataforma usada, pasamos a describir las clases implementadas y su interacción. La Figura 5.1 muestra el esquema general de funcionamiento, en el mismo podemos apreciar la interacción existente entre los experimentos a ejecutar y el API base para definir modelos. También queda especificada la alto modularidad presente gracias al uso de DESMO-J.

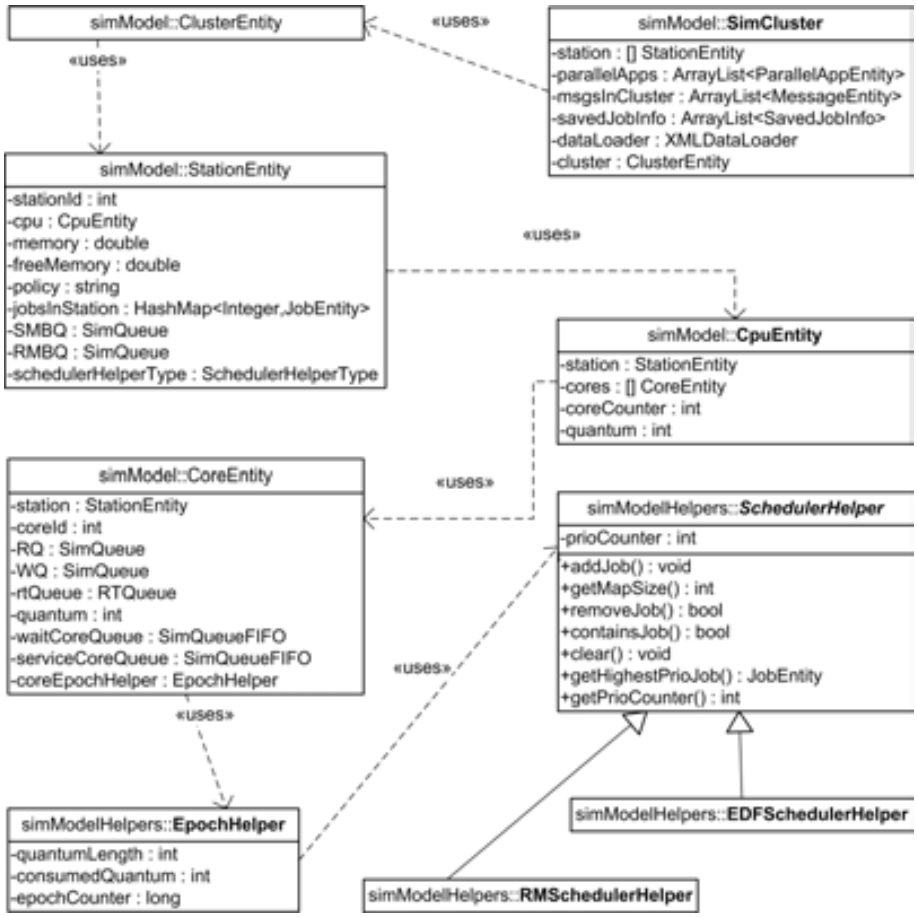

Figura 5.2: Interacción general de entidades "hardware" presentes en el modelo.

5.1.1.2. Entidades relevantes

Agruparemos las entidades presentes en el modelo en dos grupos, el primero conteniendo las entidades que representan elementos de hardware y el segundo las entidades relacionadas con los trabajos y tareas representadas.

Antes de entrar en la descripción de las entidades relevantes, queremos destacar la clase base del simulador, nombrada ***SimCluster*** y mostrada en la Figura 5.2. Esta clase es el contenedor principal de todas las entidades incluidas en el diseño y la responsable de cargar los datos de entrada, generar todos los nodos con sus respectivos estados e iniciar el proceso de simulación. El proceso de inicio de la simulación incluye la creación de una instancia de la clase *desmoj.core.simulator.Experiment*, que es la que provee la infraestructura para la ejecución de una simulación. El Algoritmo 2 muestra de forma general las principales acciones a realizar para hacer simulaciones emplean-

do el modelo definido. En este algoritmo, primero definimos las instancias y principales métodos que intervienen (líneas de la 1 a la 5) y luego mostramos las principales acciones a realizar para realizar un experimento. Con este algoritmo queremos mostrar los primeros pasos a realizar para poder ejecutar un experimento luego de definir un modelo.

Algoritmo 2 Algoritmo general de la simulación

1: SimulationModel → SimCluster Contiene las entidades y eventos del modelo
2: Experiment → Clase que provee la infraestructura para ejecutar simulaciones con los modelos definidos.
3: StopCondition → Condición de Fin, en este caso el experimento se detiene si todas las aplicaciones paralelas han terminado
4: SimCluster.**init**() → Inicializa el model, también carga los datos de *inputFile.xml*
5: SimCluster.**doInitialSchedules**() → Planifica los eventos iniciales del modelo, en este caso también crea las instancias de *StationEntity* y de *JobEntity* cargadas del fichero *inputFile.xml* y planifica los primeros eventos *ServiceStartEvent*
6: **create new** Instancia de Experiment → *experiment*
7: **create new** Instancia de SimulationModel → *simModel*
8: *simModel*.**connectToExperiment**(*experiment*)
9: **create new** Instancia de StopCondition → *stopCondition*
10: *experiment*.**stop**(*stopCondition*)
11: simModel.**init**()
12: simModel.**doInitialSchedules**()
13: *experiment*.**start**()
14: *simModel*.**printResults**(*outFile.xml*)
15: *simModel*.**finish**()

Es interesante mencionar también la manera de detener los experimentos, que se basa en condiciones de parada. Las condiciones de parada han de ser clases que hereden de la clase *desmoj.Condition* e implementen el método *check* que es el empleado para determinar si las condiciones se cumplen o no. En nuestra implementación la condición de parada es que todos los trabajos paralelos (SRT o no) cargados en el fichero de datos de entrada terminen su ejecución. Para lograr esto, mantenemos un contador de las tareas de cada trabajo paralelo que van terminando su ejecución y al ser igual este contador al valor inicial de la cantidad de tareas del nodo, aumentamos el contador de cantidad de trabajos paralelos concluidos. El método check sólo ha de comparar el valor de este contador con la cantidad inicial de trabajos paralelos incluidos y detener la simulación en caso de ser iguales.

Entidades que modelan elementos "hardware"

class ***ClusterEntity***: entidad que representa a un cluster, es el contenedor principal de todos los elementos presentes en la simulación. Mantiene la lista de los nodos pertenecientes al cluster y otras informaciones de carácter global, como una lista con todos los mensajes que están en movimiento en el cluster en cada momento.

class ***StationEntity***: entidad que representa a un nodo del cluster. Encapsula los objetos que representan los recursos del ordenador, entre ellos la CPU y sus modelos de planificación. Mantiene las siguientes colas:

- **RMBQ** (*Receive Messages Buffer Queue*): los mensajes que arriban al nodo son guardados en esta cola, a la espera de que la tarea a la cual han sido enviados entre en la CPU y los pueda procesar.
- **SMBQ** (*Send Messages Buffer Queue*): los mensajes que han de ser enviados desde este nodo son guardados en esta cola. En este implementación, los eventos de comunicación se revisan con cada evento de terminación de la CPU, por lo que en caso de generarse alguno ha de ser guardado a la espera de que la tarea que los generó entre en la CPU y los pueda procesar.

La entidad StationEntity conoce las clases que implementan las políticas de planificación de las tareas Best-effort y SRT. Como ya se mencionó en el capítulo 3, los tipos de políticas para tareas Best-effort son Round Robin y Coscheduling Cooperativo; y para SRT son RMS y EDF [88].

class ***CpuEntity***: representa la CPU de un nodo y encapsula toda la información relacionada con la misma. Cabe destacar que está incluida para crear un diseño más natural y permitir que la ampliación de las potencialidades del simulador sea más fácil.

class ***CoreEntity***: representa los cores de la CPU de un nodo y encapsula toda la información relacionada con los mismos. Los eventos *ServiceStartEvent* y *ServiceEndEvent* son los que controlan la entrada y salida de trabajos a los cores de la CPU. El control de las épocas es realizado a través de la clase *EpochHelper*, que se auxilia de las clases que implementan los algoritmos de planificación RT (RMS y EDF) para saber cuál es la próxima tarea en entrar en la CPU. Cada core tiene sus propias colas para los diferentes estados de los trabajos, las colas son:

- **RQ** (*Ready queue*): representa la cola de trabajos listos para ser ejecutados, de acuerdo a la planificación de eventos de la CPU, los trabajos pasan a la misma y luego a la cola de espera hasta que se termine la época actual. Hemos de hacer notar que en esta implementación los trabajos de tipo SRT nunca pasan a la cola de espera.

- **WQ** (*Wait queue*): contiene los trabajos que ya se han ejecutado y han de esperar a que termine la época para volver a la RQ. Cada vez que termina una época de la CPU esta cola es vaciada y todos los trabajos que hay en ella pasan a la RQ.
- **rtQueue**: en esta cola están los trabajos con características SRT.

Las relaciones entre estas entidades se observan en la Figura 5.2. *StationEntity* mantiene referencias a diferentes instancias del modelo, siendo la más interesante la instancia *CpuEntity*. Al ser creada una instancia de la clase *StationEntity*, se toma como referencia el valor de *SchedulerHelperType* para seleccionar las instancias de los *CoreEntity* asociadas a *CpuEntity*. La información del estado de la época es mantenida por *EpochHelper* en cada una de las instancias de *CoreEntity*. Cada instancia de *CoreEntity* está contenida en *CpuEntity*, y a través de ella tiene acceso a datos generales de planificación comunes a todos los cores.Todo el conocimiento del estado de la simulación es accesible desde el modelo, representado por una instancia de la clase *SimCluster*.

Entidades que modelan la carga de trabajo

class ***JobEntity***: Encapsula toda la información relacionada con una tarea. De acuerdo a nuestras necesidades, una tarea puede ser local o paralela y para cada uno de estos tipos, podemos tener características SRT. Además de los campos necesarios para controlar estas características de la tarea, hemos de llevar el control de la cantidad de jiffies consumidos por todas las tareas en cada momento, para controlar su terminación.
Al ser cargado un fichero de entrada, marcamos la entidad de acuerdo a si es local Best-effort ($LOCAL$), local SRT ($LOCAL\_SRT$), paralela Best-effort ($PARALLEL$) o paralela SRT ($PARALLEL\_SRT$). Estas marcas definen cuales campos tendrán valores, la forma en que son tratados durante la planificación de la CPU, si revisan o no las colas de mensajes (RMBQ ó SMBQ) y otros comportamientos propios de cada tipo de tarea.

class ***ParallelAppEntity***: Contiene toda la información relacionada con una aplicación paralela. A partir de los valores que la caracterizan se construyen las tareas que la conforman en los diferentes nodos.

class ***MessageEntity***: Entidad que representa a un mensaje. Conoce su nodo origen y su nodo destino, además de la tarea que lo originó. Las caracterización de las aplicaciones paralelas son tomadas en cuenta para decidir si generan o no mensajes, pues se generan de acuerdo a sus necesidades de cómputo.

La interacción de las entidades mencionadas en esta subsección con las entidades "hardware" es mostrada en la Figura 5.3. En esta figura podemos

apreciar que el conocimiento de las tareas es propio de los nodos y que en cambio información de más alto nivel se conoce desde la perspectiva del cluster. En el cluster se mantiene la información de las aplicaciones paralelas y los mensajes entre ellas, para que sea accesible a todas los nodos para su uso.

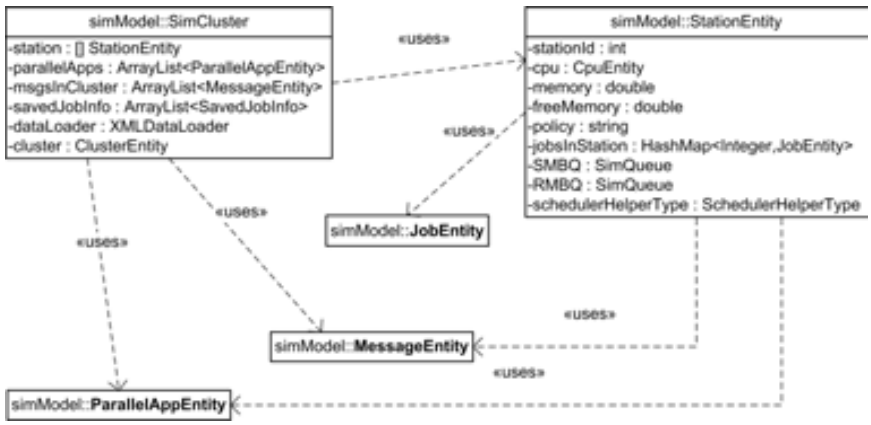

Figura 5.3: Interacción de las entidades que modelan la carga de trabajo con las entidades "hardware".

5.1.1.3. Eventos relevantes

Como se ha establecido anteriormente, los eventos controlan los cambios de estado interno, y han de estar asociados a una entidad. En esta sección describimos los eventos más importantes incluidos en este modelo, el Algoritmo 3 muestra la interacción entre ellos. Hemos desglosado este algoritmo para facilitar su comprensión, siendo el Algoritmo 5 el que explica el funcionamiento de la generación de los mensajes y el Algoritmo 4 el que describe el procedimiento seguido al finalizar una tarea.

El Algoritmo 3 muestra la interacción general de los eventos, la primera acción a realizar es cargar los datos del fichero de entrada en formato XML, crear las instancias necesarias de *JobEntity* y *StationEntity* y generar el primer evento de *ServiceStartEvent* para cada tarea (líneas 6 a 9). Es durante la creación de las instancias de *StationEntity* que creamos las instancias de *CoreEntity* asociadas a *CpuEntity* de acuerdo a los valores de las políticas en uso.

Luego para cada instancia de *JobEntity* presente en cada instancia de *StationEntity* generamos el primer evento *ServiceStartEvent*. Posterior a esto, el control de los eventos para cada instancia de *JobEntity* relacionada con las instancias de *StationEntity* pasa a los dos eventos principales del modelo, *ServiceStartEvent* y *ServiceEndEvent*. El primero realiza las acciones

necesarias (línea 13), entre las cuales está la planificación del evento *ServiceEndEvent* asociado a su ejecución. Es durante la ejecución del evento *ServiceEndEvent* (líneas 14 a 23) que se chequea la ocurrencia de eventos de RT (líneas 18 a 20), comunicación (línea 22) o terminación de tarea (líneas 15, 16).

El Algoritmo 4 ocurre cuando una tarea consume todo su tiempo de cómputo (línea 15 del Algoritmo 3). En caso de ser una tarea local, no se hace nada (línea 12). En cambio si es una tarea paralela ha de incrementarse el contador de tareas de la respectiva aplicación paralela terminadas (línea 3). De acuerdo a si esto implica que todas las tareas de la aplicación paralela han terminado o no, se incrementa el contador de aplicaciones paralelas terminadas (línea 6) o no se hace nada. Si el contador de aplicaciones paralelas terminadas es igual a la cantidad de aplicaciones cargadas del fichero de entrada, se detiene la simulación (línea 8), proceso que implica la generación del fichero de salida en formato XML .

Finalmente, el Algoritmo 5 describe el proceso de la generación de un evento de comunicación. Cabe mencionar que es un requerimiento para la ejecución de este algoritmo que la tarea sea de tipo paralelo, ya sea SRT o no. Si a tarea es paralela, procedemos a muestrear una distribución booleana de Bernoulli creada de acuerdo a la relación cómputo/comunicación de la aplicación paralela. Si el valor devuelto es verdadero, se pone en marcha el proceso de crear mensajes en los otros nodos del cluster que tienen tareas de esta aplicación paralela. Han de mencionarse que esto es solo el inicio del proceso de comunicación, que consta de más partes. Luego de ser introducidos estos mensajes en las colas de mensajes del nodo, se procesan cuando la tarea paralela recibe la CPU y se envían a los nodos a los cuales están destinados. El proceso de comunicación concluye cuando las respectivas tareas de los otros nodos con tareas cooperantes revisan sus colas de mensajes recibidos, los procesan y envían las respuestas.

class ***ServiceStartEvent***: Evento que gestiona la inserción de las tareas en los cores de la CPU, calculando el quantum que le corresponde en caso de ser necesario. Conjuntamente con el evento *ServiceEndEvent* controlan la entrada y salida de las tareas en los cores de la CPU. En caso de tener tareas SRT, asigna sus slices del quantum de la CPU de acuerdo a sus requerimientos, el slice disponible del quantum se emplea en las aplicaciones Best-effort.
Al acceder una tarea paralela la CPU, revisa las colas de mensajes para saber si han ocurrido eventos de comunicación. En caso de ser así, se gestionan en ese momento.

class ***ServiceEndEvent***: cada vez que ocurre un evento *ServiceStartEvent*, se genera un evento de este tipo, que se encarga de expulsar la tarea de la CPU. Al ocurrir este evento se actualizan todas las variables que controlan las cantidades de CPU recibidas por cada tarea, en caso de terminar la tarea,

Algoritmo 3 Interacción general entre los eventos en el modelo

1: **ServiceStartEvent** → Evento que ocurre cada vez que una tarea necesita la CPU, durante su ejecución dispara un evento **ServiceEndEvent**
2: **ServiceEndEvent** → Evento que ocurre cada vez que una tarea deja la CPU, puede generar: *Fin de Tarea*, *Condición RT* y *Comunicación*.
3: **JobArrivalEvent** → Ocurre cada vez que una tarea arriba al sistema, durante la carga de datos del fichero de entrada.
4: **CommunicationEvent** → Representa un evento de comunicación, implica que se revisen las colas locales de mensajes.
5: **RTEvent** → Representa un evento de RT.
6: **for all** $Job \in inputFile.xml$ **do**
7: Generar la correspondiente instancia de ($JobEntity \rightarrow job_{i,j}$) del Job_i en $station_j$
8: Asignar el trabajo $job_{i,j}$ en $station_j$ a uno de los cores de la CPU acuerdo a la política (BY_APP, BY_USR, etc) en uso
9: Planifica para $job_{i,j}$, su primer evento **ServiceStartEvent** en $node_j$, en orden de prioridad para las tareas SRT y las Best-effort luego.
10: **end for**
11: **for** $j = 0$ **to** $simCluster.getCantStations()$ **do**
12: **for** $i = 0$ **to** $station[j].getCantJobs()$ **do**
13: **while** ($job_{i,j}.remCV > 0$) **do**
14: Durante el evento **ServiceStartEvent** de cada CPU core:
Planificación de un evento **ServiceEndEvent** en $t = t_{actual} + job_{i,j}.assignedQuantumSlice$
Chequeo de las colas de mensajes, $SMBQ$ y $RMBQ$ (envío y recepción de mensajes).
Control de Época de la CPU y del quantum a asignar a las tareas
15: Durante el evento **ServiceEndEvent**:
Cálculo del tiempo de cómputo restante para la tarea → $job_{i,j}.remCV$
16: **if** ($job_{i,j}.remCV == 0$) **then**
17: Ejecutar **Algoritmo 4** (Fin de Tarea)
18: **else**
19: **if** ($job_{i,j}.RT$) **then**
20: Generar evento **RTEvent** → $rtEvent$
21: Planificar $rtEvent$ para $t = t + job.getPeriod()$
22: **end if**
23: Ejecutar **Algoritmo 5** (Generación de Comunicación)
24: **end if**
25: **end while**
26: **end for**
27: **end for**

Algoritmo 4 Fin de Tarea

```
1: La instancia de JobEntity, job ha consumido todo su volumen de cómputo
2: if ((job.type == PARALLEL) or (job.type == PARALLEL_SRT)) then
3:    Incrementar el contador de tareas terminadas (finishedJobs) de la
      aplicación paralela correspondiente.
4:    if (finishedJobs == parallelAppJobsCounter) then
5:       Guardar la información de la aplicación paralela para generar el
         fichero de salida (outFile.xml)
6:       Incrementar el contador de tareas paralelas terminadas finishedPa-
         rallelApps
7:       if (finishedParallelApps == loadedParallelApps) then
8:          Condición de Finalización alcanzada, detener la simulación y ge-
            nerar el fichero de salida (outFile.xml)
9:       end if
10:   end if
11: else
12:   Fin de Tarea Local → Se descarta su información
13: end if
```

se trata de acuerdo a si es local o paralela. En caso de ser local, nada ocurre, solo se guardan sus datos y se borra de las colas del nodo. Por otro lado, en caso de ser paralela también se quita de las colas del nodo y además se actualiza el contador de tareas en el trabajo paralelo. Si este contador vale 0, se notifica el fin del trabajo y se guarda hasta la terminación de la simulación.

Algoritmo 5 Generación de Evento de Comunicación

```
Require: (job.type == PARALLEL) or (job.type == PARALLEL_SRT)
1: Muestrear la distribución de Bernoulli (creada de acuerdo a la relación
   cómputo/comunicación) de la tarea paralela para decidir si comunica o
   no → generateCommEvent
2: if (generateCommEvent) then
3:    for (i = 0 to simCluster.stationCounter) do
4:       if (stationHasJob(job, i) and (i ≠station.getStationId())) then
5:          create new MessageEntity → msg
6:          Adicionar msg a la cola de mensajes en el cluster
7:          station.SMBQ.insert(msg);
8:       end if
9:    end for
10: end if
```

Los datos de las aplicaciones paralelas conforman un fichero de salida en formato XML, que recibe LoRaS y constituyen los valores de $RExT$ generados por Simulador_Cluster_SRT.
En este evento se gestionan los eventos de tiempo real, que es descrito a continuación.

class ***RealTimeEvent***: representa a un evento RT, es planificado solo en dos situaciones:
1– Al arribar una tarea con características SRT al sistema.
2– Cuando expulsamos alguna tarea SRT de la CPU.
Al ocurrir, desaloja (si no es SRT) el trabajo que se encuentra en la CPU y se planifica un evento ServiceStartEvent.

class ***JobArrivalEvent***: Debido a que los datos de los trabajos son cargados de ficheros de entrada, el peso de los datos recae en las tareas. Para cada nodo recibimos una lista de las tareas presentes en él, de cualquier tipo y con cualesquiera características (SRT o Best-effort). A partir de esta lista, se cargan los datos y el modelo se inicializa con ellos, este evento gestiona la colocación de cada tarea en el nodo que le corresponde, y la correcta selección del core de acuerdo a la política de asignación en uso.

class ***CommunicationEvent***: Representa un evento de comunicación, al ocurrir, se revisan las colas de mensajes del nodo y los mensajes pertenecientes a la tarea se procesan.

5.1.2. Soporte para algoritmos RT

Para facilitar la experimentación de nuevos algoritmos de planificación RT, se diseñó el grupo de clases que conforman esta parte del software de forma modular. Esta implementación, permite añadir nuevos algoritmos heredando de una clase abstracta, nombrada *SchedulerHelper* e incluida en el paquete *simModelHelpers*. La Figura 5.4 muestra las clases que representa a los algoritmos inicialmente incluidos en la jerarquía (*RMSchedulerHelper* y *EDFSchedulerHelper*) y la interacción entre las clases necesarias para realizar la planificación.

La clase *StationEntity* es el contenedor principal de toda la información de las tareas que se encuentran en ella y la forma de planificarlas. Por esta razón guarda referencias a los *helpers* para la planificación usando algoritmos RT y a la CPU. La clase *CpuEntiy* se apoya de la información brindada por la clase *CpuEpochHelper* (también definida en el paquete *simModelHelpers*) a la hora de controlar sus épocas. Esta a su vez, se nutre de la información que brindan las clases mostradas en la Figura 5.4 (*RMSSchedulerHelper* o *EDFSchedulerHelper*) para decidir datos relacionados con el quantum o cual es el la próxima tarea RT en entrar en la CPU. Implementando esta parte del software de esta forma logramos una mayor extensibilidad y flexibilidad

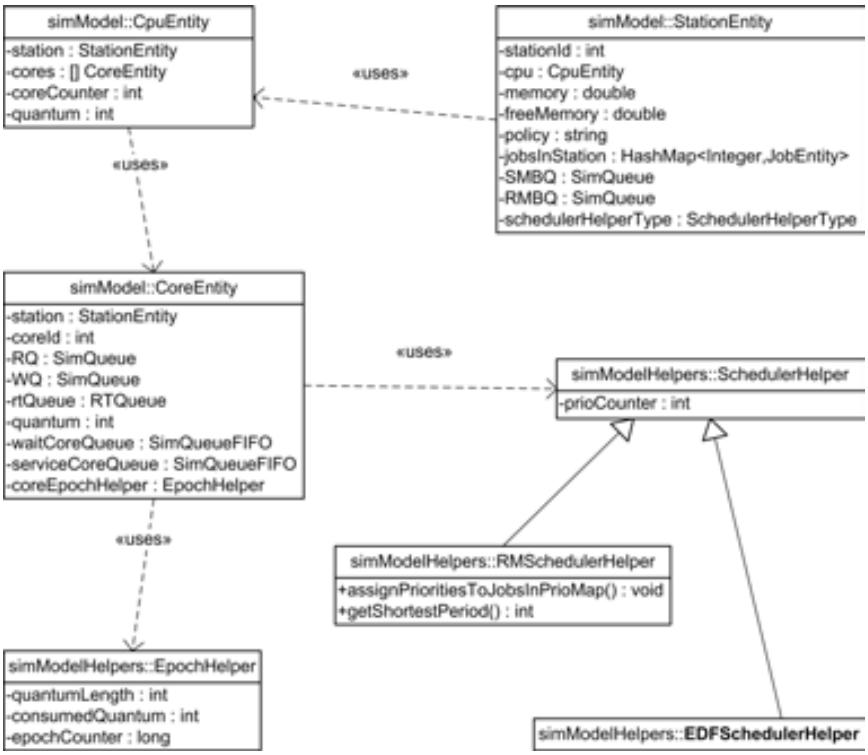

Figura 5.4: Interacción de las clases, soporte para adición de algoritmos de planificación RT.

en el código, que se traduce en ahorro de tiempo y esfuerzo para desarrollos futuros.

5.1.3. Soporte para procesadores multi-core

Dotar a nuestra herramienta de soporte para procesadores multi-core es uno de nuestros objetivos fundamentales, tanto a nivel de simulación como para las ejecuciones reales. En el caso actual, la herramienta de simulación, hemos reproducido en nuestro diseño el orden existente en los procesadores reales, con el fin de facilitar la inclusión de más potencial al simulador en un futuro, como por ejemplo las cache.

Para lograr que nuestra herramienta pueda simular procesadores multi-core, hemos tenido que migrar parte de las responsabilidades de diferentes entidades a la nueva entidad para representar a los cores, llamada *CoreEntity*. La Figura 5.5 muestra a la izquierda la interrelación entre las diferentes entidades que representan a los elementos hardware en el modelo en la actualidad, y a la derecha el diseño original. Los detalles más llamativos de las modificaciones para soportar multi-core son la creación de la entidad

CoreEntity y la migración de las colas de tareas a los a dicha entidad. Como cada core tiene su propia información de planificación, dispone de sus propias instancias de *EpochHelper* y *SchedulerHelper*. Solo queda mencionar que las instancias de EpochHelper se crean en base al valor del campo *schedulerHelperType* de *StationEntity*.

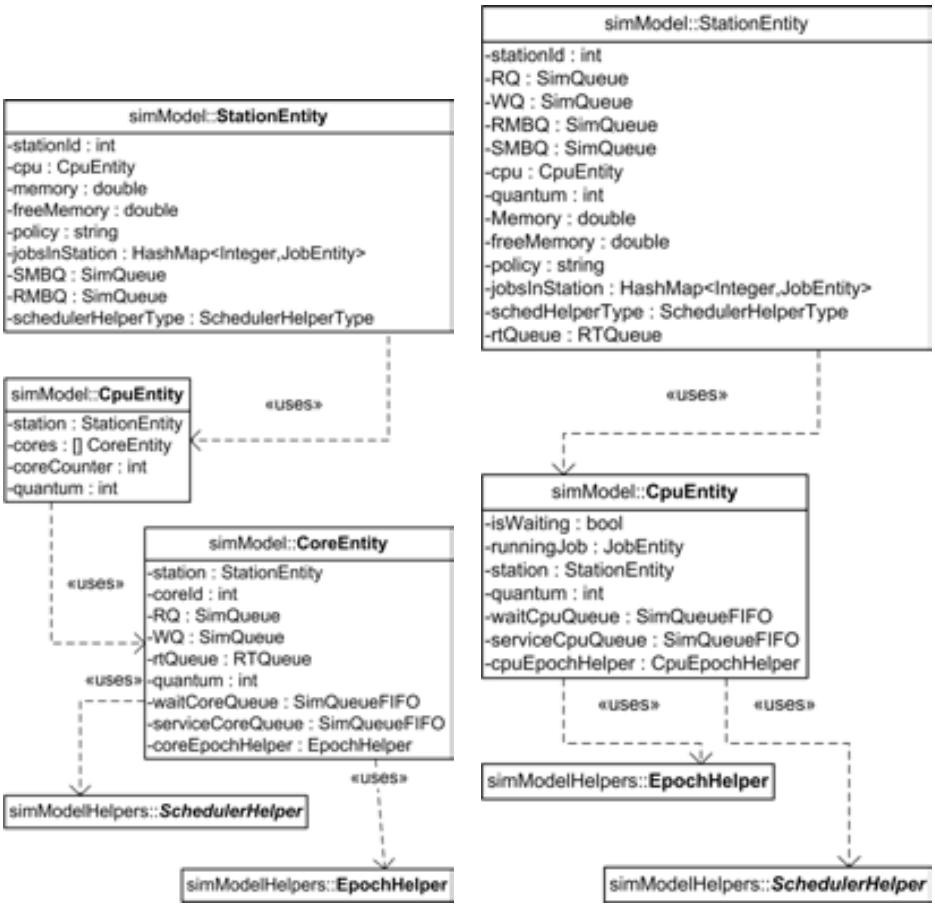

Figura 5.5: Comparación de entidades hardware con y sin soporte para procesadores multi-core.

5.1.4. Detalles de la implementación de la comunicación entre los simuladores

Cabe destacar que, al igual que con el soporte de nuevos algoritmos, la modularidad del código implica mayor facilidad a la hora de extender las funcionalidades a nuevos tipos de tareas.

Figura 5.6: Jerarquía de manejo de datos y su interacción general.

Por las razones antes expuestas, ahorra tiempo de desarrollo y hace más claros y legibles los ficheros de intercambio de datos. Es también un estándar ampliamente utilizado.

5.2. CISNE_SRT

Entendemos por CISNE_SRT a la unión del entorno de planificación CISNE y SRT_Scheduler, el planificador que hemos diseñado en este trabajo para administrar el potencial multi-core y planificar la mezcla de tareas que da origen a este trabajo. Más datos sobre el diseño e implementación de CISNE pueden ser consultados en la tesis doctoral del Sr. M. Hanzich [49].

5.2.1. Modificaciones realizadas

En esta sección mencionaremos una serie de modificaciones ligeras y adiciones realizadas al entorno CISNE para permitir su interacción con los subsistemas adicionados al entorno: SRT_Scheduler y Simulador_Cluster_SRT.

Adiciones

- Módulo de gestión de ficheros XML, necesario para la comunicación entre LoRaS y Simulador_Cluster_SRT.

Modificaciones

- Modificaciones al proceso de lectura de ficheros de configuración, con el objetivo de introducir los datos relacionados con los nuevos tipos de aplicaciones, los procesadores multi-core y las políticas de asignación de cores.
- Revisión del código relacionado con las librerías p_thread, que presentaba incompatibilidades con la versión 2.6 del kernel de Linux, causando que los demonios de LoRaSd quedasen en estado zombie al lanzarse.
- Revisión de los módulos de comunicaciones, que al ser portado al kernel 2.6 también presentó problemas en la comunicación a través del protocolo TCP/IP entre el nodo servidor y las estaciones de trabajo incluidas en el fichero de nodos empleado por el entorno.

5.2.2. SRT_Scheduler

SRT_Scheduler es como hemos denominado a nuestra propuesta para incluir en el entorno un nuevo nivel de planificación temporal, que actúe de intermediario entre el entorno y el sistema operativo y que se ocupe de administrar las características distintivas de los procesadores multi-core. En esta sección mencionamos el diseño de clases implementado en SRT_Scheduler, y la interacción entre ellas.

5.2.2.1. Arquitectura

Como hemos mencionado antes, SRT_Scheduler es un software independiente, accesible a los otros componentes del entorno a través de comandos. La Figura 5.7 muestra la ubicación de los módulos de LoRaS en un cluster, y el posicionamiento como intermediario de nuestro middleware. Como parte de SRT_Scheduler se instala un ejecutable parametrizable capaz de enviar los

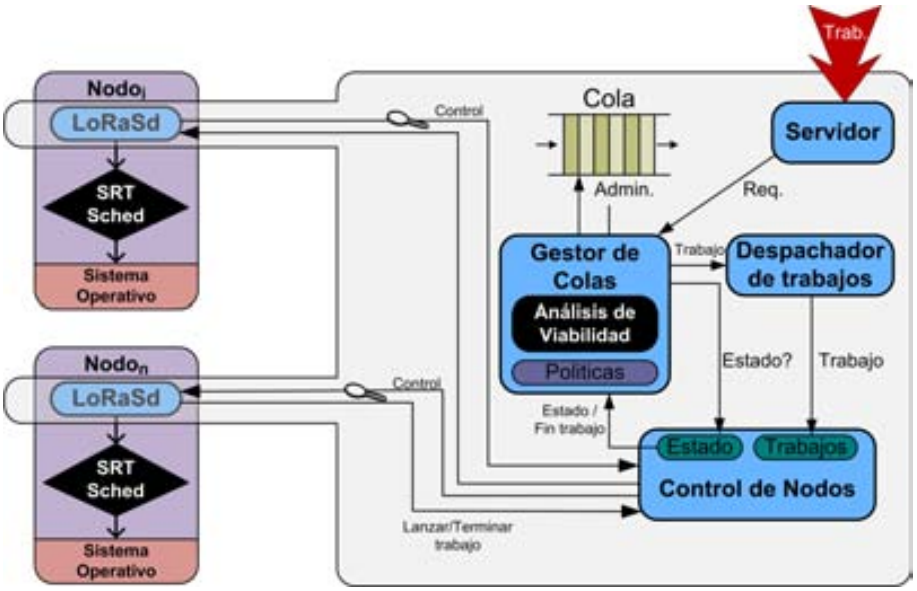

Figura 5.7: Módulos de LoRaS en un cluster, interacción con SRT_Scheduler.

comandos al demonio principal, de esta forma podemos acceder fácilmente a las potencialidades multi-core tanto a nivel local, desde el demonio de LoRaS, como desde el nivel superior de LoRaS.

El sistema de recolección de información de LoRaS, a través de sus demonios por nodo, combinado con el conocimiento del estado de cada nodo del módulo Gestor de Colas, nos permite realizar un análisis de viabilidad basado en la memoria principal y la CPU. Nuestro demonio de control monitoriza el uso de recursos de forma periódica, y envía esta información al nodo servidor para su uso por LoRaS, que además mantiene un registro de las aplicaciones paralelas por nodo.

5.2.2.2. Clases relevantes

La Figura 5.8 muestra las principales relaciones de uso en nuestro planificador SRT. Al ser el punto de entrada TSRTScheduler ha de tener referencias a cada una de las clases que implementan las funcionalidades presentes en el planificador, como por ejemplo el módulo de comunicaciones o el de cambio de afinidad. Nótese que las clases TSRTAlgorithm y TSchedulingPolicy son metaclases, lo que significa que pueden contener instancias de sus subclases.

La clase *TCommunicationsManager* se encarga de proveer la infraestructura para las comunicaciones, hemos de recordar que nuestro planificador recibe comandos del nivel superior, en este caso LoRaS.

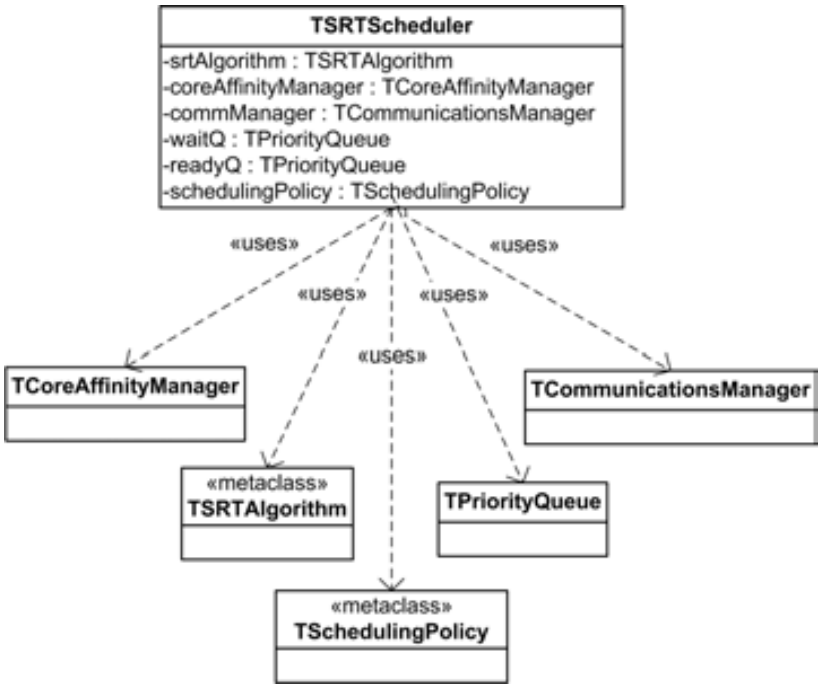

Figura 5.8: Relaciones de uso en SRT_Scheduler.

La clase *TCoreAffinityManager* se encarga de proveer todos los elementos necesarios para el control de la afinidad en el procesador.

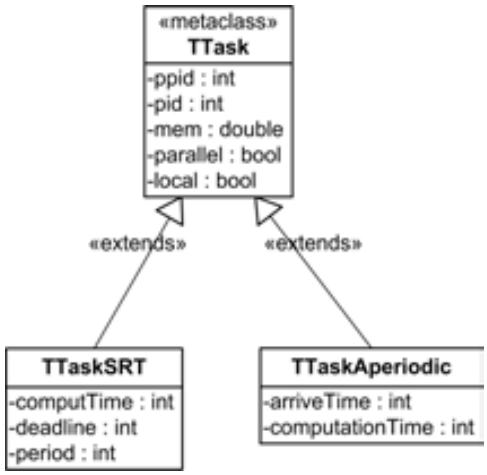

Figura 5.9: Jerarquía de clases que representan las tareas.

La clase *TPriorityQueue* es nuestra implementación de una cola de prioridades para el uso del planificador.

La clase *TSRTAlgorithm* es la clase abstracta base de la jerarquía de posibles algoritmos de planificación a emplear por nuestro planificador para las tareas SRT, en la actualidad posee solo una subclase *TSRTAlgorithmRMS*, que implementa nuestra versión del algoritmo RMS. La Figura 5.9 contiene la jerarquía actual de los tipos de tarea que puede planificar nuestro planificador, nótese que existe una clase abstracta, *TTask*, que contiene el comportamiento compartido por sus clases hijas.

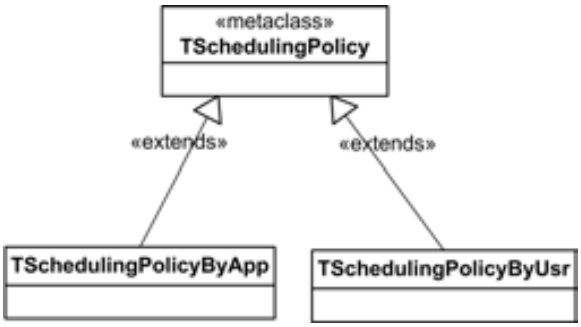

Figura 5.10: Jerarquía de clases que representan las tareas.

La clase *TSchedulingPolicy* es la base de la jerarquía de políticas de asignación de cores, en la actualidad tiene dos clases que la implementan, *TSchedulingPolicyByApp* y *TSchedulingPolicyUsr*, como se muestra en la Figura 5.10. Estas clases contienen los elementos necesarios para implementar nuestra políticas de asignación de cores.

Tarea		SRT_Scheduler	Linux
Planificador		0	100-105
local	SRT	1	106-113
paralelo	SRT	2	114-123
local	Best-effort	3	124-131
paralelo	Best-effort	4	132-139

Tabla 5.1: Mapeo de prioridades de Linux y SRT_Scheduler.

5.2.2.3. Sistema de prioridades

SRT_Scheduler está desarrollado totalmente en espacio de usuario, con las limitaciones que esto conlleva. Por esta razón hemos implementado nuestro control de prioridades a través de las funciones para el control de prioridades en Linux accesibles a nivel de espacio de usuario, que controlan las prioridades estáticas del sistema operativo. El control de prioridades se encuentra en el módulo principal de nuestro sistema.

Una de las características de las prioridades en Linux es que se calculan en base a la prioridad fija o estática, que es la que somos capaces de controlar, y un sistema de bonos y prioridades dinámicas basado en el comportamiento del consumo de CPU de la aplicación y su tiempo de espera en el estado de *wait*. Para evitar en lo posible que las prioridades cambien como resultado de este comportamiento de Linux, mapeamos un rango de prioridades de Linux a cada una de las prioridades internas de nuestro planificador, de la forma mostrada en la Tabla 5.1.

Capítulo 6

Experimentación Realizada y Resultados Obtenidos

En este capítulo mostramos la experimentación realizada, inicialmente nos centraremos en las caracterizaciones de los entornos y las aplicaciones, tanto para el entorno real como para el simulado. Posteriormente pasaremos a mostrar la experimentación relacionada con la simulación, principalmente dirigida a validar el sistema y observar el desempeño de nuestro método de estimación simulado ante diferentes situaciones. Finalmente mostraremos los datos relacionados con las ejecuciones reales, prestando especial atención a la coplanificación para entornos multicore.

6.1. Caracterización de los entornos de ejecución

La caracterización de los entornos de ejecución es muy importante para nuestro trabajo, pues ubica la experimentación realizada y nos permite mostrar las bases en las que se cimenta nuestro trabajo.

6.1.1. Particularidades del escenario bajo estudio

Creemos que es de especial importancia la correcta contextualización de nuestro trabajo, con el fin de facilitar el análisis de los resultados alcanzados. Con este fin hemos creado esta subsección, dónde describiremos nuestro escenario, tomando en cuenta tanto las particularidades que lo hacen especial, como las características de los benchmarks empleados y la carga local presente en este entorno.

Las NOWs o clusters no dedicados suelen componerse de grupos de ordenadores pertenecientes a instituciones, que desean sacar un provecho extra

de su parque de ordenadores, haciendo coexistir las tareas de las aplicaciones paralelas y locales en los nodos que lo conforman. Este tipo de enfoque ha de respetar la presencia del usuario local, es decir, los nodos que conforman nuestras NOWs son ordenadores en uso por miembros de las instituciones que crean; siendo de especial importancia mantener los niveles de capacidad de respuesta. En este trabajo nos referimos a los miembros de las instituciones como *usuarios locales*, y el entorno de planificación está creado de tal forma que respeta lo que se denomina el Contrato Social. El Contrato Social establece los límites de recursos disponibles para las aplicaciones paralelas, y se establece de tal forma que los usuarios locales no se vean penalizados de forma perceptible para ellos.

Una vez establecido este punto, pasamos a otra particularidad importante de nuestro escenario, consideramos que nuestras NOWs están formadas por los ordenadores de los laboratorios de clases de las universidades. Esta restricción es muy importante, pues acota los tipos de aplicaciones que puede lanzar el usuario local y permite establecer con cierto nivel de seguridad los consumos de recursos que tendrán.

La carga impuesta al entorno está descrita en detalle en la sección 6.2, basándonos en los datos obtenidos por nuestro entorno de planificación. No obstante, es necesario aclarar algunos puntos importantes relacionados con los benchmarks empleados en este trabajo:

1. El arribo de las aplicaciones para cargas de trabajo mayores de 20 ha sido creado usando una distribución de *Poisson* [37].

2. Es una práctica común en los clusters, dedicados o no dedicados, pedir información al usuario paralelo sobre la aplicación que lanza, siendo bastante normal que la información que proveen sea incorrecta. Para evitar esta situación en la medida de lo posible, el diseño de CISNE incluye un módulo de Caracterización de Aplicaciones , la información que recaba este módulo nos sirve para diversos fines descritos en la memoria.

6.1.2. Entorno de las ejecuciones reales

La experimentación mostrada en este trabajo fue realizada en un cluster de 8 nodos, con la configuración de hardware mostrada en la Tabla 6.1. Como podemos comprobar en dicha tabla, nuestro cluster es homogéneo y compuesto de nodos con procesadores de dos cores. Esta limitación ha condicionado nuestro trabajo en lo relacionado con los experimentos que involucran ejecuciones reales, siendo la razón por la que nuestras políticas se centran en distribuir las tareas pertenecientes los diferentes tipos de aplicaciones solo entre dos cores.

Componente	Valor	Observaciones
Procesador	Pentium-D	3.40 GHz
L2	2x2 MB	No compartida
Cores	2	
RAM	2x512 MB	Dual channel
HDD	80 GB	SATA
LAN	100 Mbps	

Tabla 6.1: Caracterización del hardware de los nodos del cluster de pruebas.

El sistema operativo en uso durante los experimentos ha sido la distribución CentOS de Linux, con el kernel 2.6.7; del administrador de recursos PBS instalamos la versión open-source, la OpenPBS 2.3.16 y la versión de MPI instalada es la openmpi 1.2.7.

6.1.3. Entorno de las ejecuciones simuladas

Para crear el entorno de simulación, intentamos reproducir lo mejor posible los datos reales del entorno previamente descrito. Con este fin los datos mencionados en la Tabla 6.1 han sido introducidos en los ficheros de entrada necesarios para el modo de simulación del entorno. El simulador también necesita los datos relativos a las aplicaciones, tanto las locales como las paralelas, introducidos en los ficheros de entradas de la carga de trabajo. Estos datos son enumerados y analizados en la sección 6.2. Además de esta información el entorno también necesita un fichero con los nombres de las aplicaciones y sus tiempos de arribo al sistema, estos tiempos de arribo son generados de acuerdo a una distribución de Poisson.

6.2. Caracterización de la carga

Debido al funcionamiento de nuestro esquema de ejecuciones reales y simuladas, disponer de una caracterización de la carga fiable y completa es de vital importancia. En esta sección mostraremos los valores obtenidos para la caracterización tanto de las aplicaciones locales como de las paralelas. En el caso de las aplicaciones paralelas hemos obtenido los valores a través del módulo de Caracterización de Aplicaciones, que es parte del prototipo para las ejecuciones reales. Para caracterizar las aplicaciones locales hemos recurrido a la medición del consumo de recurso de las mismas. En las siguientes subsecciones mostramos los resultados obtenidos.

6.2.1. Aplicaciones locales

Para representar el uso de recursos de las aplicaciones locales en los nodos de nuestro cluster hemos creado un benchmark parametrizable, que es capaz de emular la presencia del usuario local. Dado que en este trabajo consideramos aplicaciones locales de dos tipos, Best-effort y soft real-time (SRT), existen versiones del benchmark para ambos tipos de aplicaciones locales. A continuación enumeramos los valores específicos para cada benchmark.

Para el caso de las aplicaciones locales Best-effort, los valores de estos parámetros han sido tomados de mediciones realizadas en laboratorios de la Universidad de Lérida [98] y con los valores obtenidos creamos los parámetros modelo para aplicaciones Best-effort. En este caso, los valores para el benchmark son 15 % de CPU, 35 % de memoria y un consumo de red de 0,5 KB/sec. Aunque estas mediciones no están totalmente actualizadas, representan una cota superior de los posibles consumos de recursos, pues el tipo de aplicación medida no ha aumentado considerablemente su consumo. También es de considerar que los recursos de hardware disponibles si han crecido en calidad y cantidad, tanto en capacidad de memoria principal, almacenamiento, poder de cómputo e interfaz de red.

k	**c**	**n**	**ram**	**ex**	**comp**	**j_aplic**	**j_total**
ep	C	4	4	186	100	18845	18854
lu	B	4	51	196	83	16539	19847
is	C	4	387	126	39	5086	13151
bt	B	4	276	326	84	27553	32745
ep	C	8	4	126	100	13415	13417
lu	B	8	30	136	81	11198	13749
is	C	8	196	116	36	4078	11416

Tabla 6.2: Caracterización de las aplicaciones paralelas MPI-NAS.

Para el caso de las aplicaciones SRT las mediciones han sido realizadas en la configuración de hardware en uso actualmente, tomando como base el reproductor multimedia Xine [2]. Esta aplicación tiene distribuciones para GNU Linux, FreeBSD, Solaris y otras; en nuestro caso empleamos la distribución para GNU Linux. Al ejecutar Xine con un tamaño de ventana de visualización de 1:1, lo cual significa que la ventana donde se muestra el vídeo es del mismo tamaño que el fichero a visualizar, los valores obtenidos de consumo de CPU variaban entre el 17 y el 20 %, consumiendo el 5.1 % de la memoria principal (51 MB).

6.2.2. Aplicaciones paralelas

En el caso de las ejecuciones reales, es necesario simular la presencia de usuarios locales y además, aplicaciones paralelas que lleguen al sistema en intervalos de tiempo representativos. Por estas razones, las aplicaciones han de estar correctamente caracterizadas, para garantizar luego que las comparaciones con el método simulado sea justa. En este trabajo hemos seleccionado los benchmarks NAS [17], por la flexibilidad que brindan a la hora de seleccionar durante los experimentos si las aplicaciones serán *CPU bound* o *IO bound*. Nosotros consideramos una aplicación *CPU bound* si el porcentaje cómputo-comunicación, calculado en base a los valores de la caracterización de la misma, es mayor del 50 %. De la misma manera, una aplicación es considerada *IO bound* si dicho porcentaje es menor que el 50 %.

K	**Nombre del núcleo**	**Descripción**
ep	Embarrassingly parallel	Prueba el rendimiento sin comunicación significativa entre los procesadores
is	Integer sort	Ordenamiento de enteros, pone a prueba tanto el poder de cómputo como la comunicación
lu	LU solver	Representa los cálculos asociados con un nuevo tipo de algoritmos CDF implícitos
bt	Block tridiagonal solver	Otros tipos de algoritmos CDF en uso por el centro Ames de la NASA

Tabla 6.3: Valores posibles de los kernels de las aplicaciones NAS y sus significados.

La carga paralela está representada por una lista de los benchmarks NAS lanzadas empleando Open MPI [47], que emplean 2, 4 y 8 nodos y llegan al sistema siguiendo una distribución de Poisson. Dichos benchmarks han sido mezclados de tal forma que estén balanceados en cuanto a cómputo y comunicación. La Tabla 6.2 muestra la caracterización de los benchmarks del NAS utilizados. Las cabeceras de la Tabla 6.2 son las siguientes:

- **k**: Kernel del benchmark NAS, puede tener los siguientes valores resumidos en la Tabla 6.3.
- **c**: Clase del benchmark del NAS, para nuestra finalidad basta con saber que a la letra A le corresponde el menor tamaño, y que se incrementa con la clase B y así sucesivamente. De cada kernel del NAS hemos seleccionado la mayor clase que puede lanzarse en nuestra configuración de hardware que evita la paginación y respeta el Contrato Social.
- **n**: Cantidad de procesadores empleados disponibles para el benchmark.

- **ram**: Cantidad de memoria principal que ocupa el benchmark del NAS, medido en megabytes (MB).
- **ex**: Tiempo de ejecución de caracterización del benchmark, lanzado en la cantidad de nodos con dedicación exclusiva que precise. Este valor está medido en segundos.
- **comp**: Porcentaje de CPU requerido por el benchmark.
- **j_aplic**: Cantidad de jiffies consumidos por la aplicación, presumiblemente el cómputo realizado.
- **j_total**: Cantidad de jiffies consumidos en total por la aplicación, la diferencia entre este valor y j_aplic nos permite calcular el tiempo gastado por aplicación en tareas de comunicación.

La mera observación de estos valores de caracterización nos permite notar que los resultados de la caracterización siguen las pautas marcadas en la literatura [108], en relación al porcentaje Cómputo/Comunicación de los benchmarks del NAS para MPI. También salta a la vista que para iguales valores de kernel y clase, en consumos de recurso disminuye al aumentar la cantidad de procesadores, resultado por demás bastante lógico.

Nuestra revisión del estado de arte nos permitió encontrar aplicaciones paralelas con características SRT referenciadas en la literatura, como lo son [110, 85], siendo esta una de las motivaciones de este trabajo. Para este trabajo construimos las aplicaciones paralelas SRT en base a las caracterizaciones de las aplicaciones NAS-MPI disponibles, empleando la Ecuación 6.1.

$$deadline(j) = 2 \times turnaround_{isol}(j) \tag{6.1}$$

En este enfoque, las aplicaciones paralelas SRT son aplicaciones paralelas Best-effort a las que se les definimos un deadline o tiempo de finalización máximo. Cabe destacar que este deadline lo calculamos multiplicando por dos el tiempo de ejecución de caracterización de la aplicación paralela, la Tabla 6.4 muestra un resumen del estudio llevado a cabo para emplear este factor de incremento.

Los factores tomados en cuenta para este estudio son la presencia de usuario local (Carga Loc.), el grado de multiprogramación (MPL) y en cuanto hemos aumentado el tiempo de caracterización (Factor). En todos los experimentos con presencia de usuario local Best-effort todos los nodos son afectados por dicha carga, los valores son los mencionados en la sección 6.2.1. Los resultados se muestran por política, y están creados a partir de tres ejecuciones de la carga Wkld para cada combinación de Factor, MPL y Carga Loc. La carga

Wkld	Factor	MPL	Carga Loc.	LIN++	BY_APP	BY_USR
8 (25 % SRT)	1.25	3	sin	0	0.67	0.33
			Best-effort	0	0.33	0
		2	sin	1	1	1
			Best-effort	1	1	1
	1	3	sin	0	0	0
			Best-effort	0	0	0
		2	sin	0.67	1	1
			Best-effort	0.67	1	1
	0.50	3	sin	0	0	0
			Best-effort	0	0	0
		2	sin	0.67	0.67	1
			Best-effort	0	0.67	0.33
	0.25	3	sin	0	0	0
			Best-effort	0	0	0
		2	sin	0	0	0
			Best-effort	0	0	0

Tabla 6.4: Estudio del factor de incremento empleado para calcular el deadline.

Wkld está formada por 8 benchmarks del NAS, de las cuales 2 son de tipo SRT, que arriban aleatoriamente al sistema. El valor mostrado para cada política tiene 4 posibles valores, 0 en caso de que ninguno de los benchmarks con deadline definido terminen antes del mismo para todas las ejecuciones, 0,33 si lo cumplen para 1, y así sucesivamente.

Observando estos datos podemos comprobar que para un $MPL = 3$, hemos de aumentar el tiempo de caracterización en 1,25 veces para lograr que se cumplan algunos deadlines. En cambio para un $MPL = 2$ comenzamos a obtener buenos resultados para un factor de incremento de 0,50, en dependencia de si el experimento se realiza con o sin carga local. Para los datos mostrados la mejor combinación de MPL y factor de incremento es la lograda al fijar dichos valores en 2 y 1 respectivamente, pues la presencia de carga local no afecta el cumplimiento de los deadlines de los benchmarks marcados como SRT para las políticas propuestas en este trabajo.

6.2.3. Métricas

La métrica empleada en este trabajo es el tiempo de turnaround de las aplicaciones paralelas, que se define como el tiempo que transcurre desde que el usuario entrega la aplicación paralela al entorno de planificación hasta que recibe los resultados de su lanzamiento. Lógicamente este tiempo com-

prende tanto el tiempo de espera en las colas del entorno, como su tiempo de ejecución real.

6.3. Comparación del entorno con PBS

Hemos incluido esta sección con el fin de establecer una pauta de comparación entre nuestro entorno de planificación y otros administradores de recursos en NOWs, más conocidos y aceptados por la comunidad científica. El entorno de planificación seleccionado es **PBS**, un gestor de colas con una versión *open-source* muy legalizada y ampliamente conocido y aceptado en la planificación de aplicaciones para NOWs. La configuración de PBS empleada durante esta experimentación es la descrita en la sección 2.3. Para este experimento hemos empleado la política LIN , que hace balance de carga y control del MPL sin emplear afinidad de cores.

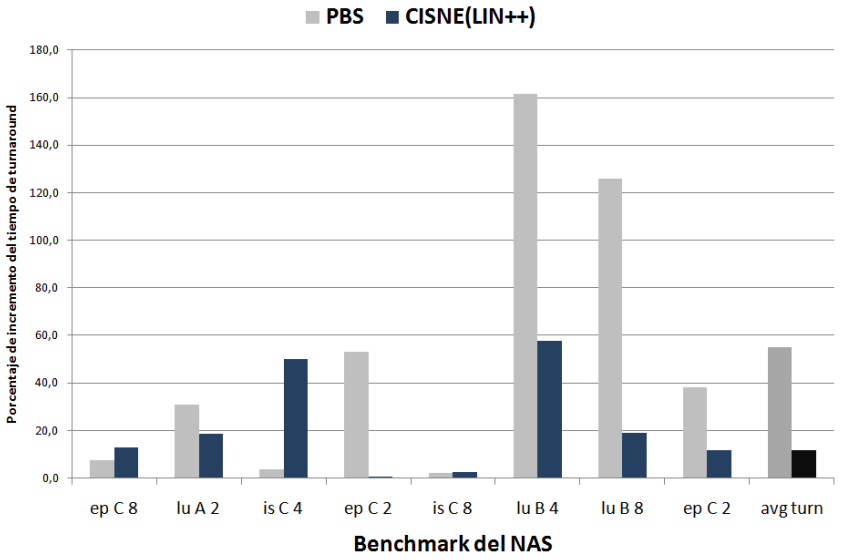

Figura 6.1: Porcentaje de incremento en los tiempos de ejecución de PBS y CISNE, comparados contra los tiempos ideales de ejecución.

El entorno PBS no permite especificar aplicaciones SRT, hemos decidido emplear una carga de trabajo que no las incluya, compuesta de varios tipos de benchmarks del NAS que arrivan al sistema siguiendo una distribución de Poisson. La Figura 6.1 muestra los porcentajes de incremento desglosados por cada aplicación. En la figura los resultados etiquetados como *avg turn* representan el turnaround promedio para ambos entornos, que muestra una clara diferencia. El incremento promedio del tiempo de turnaround contra las ejecuciones ideales para PBS es de casi un 55%, mientras que para nuestro

entorno es del 11,6 %. Hemos empleado los porcentajes de incremento en vez de la comparación directa entre los tiempos de turnaround para evitar que si se afectase la ejecución de una aplicación paralela con un tiempo de ejecución muy largo los resultados fuesen engañosos. Cabe destacar que una comparación del turnaround promedio de CISNE y PBS para esta carga arroja que el resultado de CISNE es un 28 % mejor que el de PBS.

% CPU	Arr	Nombre	Ex Ideal	PBS	CISNE	%Inc PBS	%Inc CISNE
100	1	epC8	126	135	142	7,5	12,7
85	58	luA2	92	120	109	30,8	18,5
40	114	isC4	126	131	189	3,8	50,0
100	180	epC2	426	653	429	53,2	0,7
40	240	isC8	116	118	119	2,0	2,6
85	274	luB4	196	513	309	161,5	57,7
85	318	luB8	136	307	162	126,1	19,1
100	380	epC2	426	588	376	38,1	11,7
			206	318	229	54,9	11,6

Tabla 6.5: Información adicional sobre la carga.

Para facilitar el análisis de la Figura 6.1 aportamos también los datos de la Tabla 6.5. En esta tabla las columnas contienen el porcentaje de uso de la CPU (% CPU), el momento en que la aplicación paralela arriba al sistema (Arr), el nombre (Nombre), los tiempos de ejecución para la caracterización (Ex Ideal), PBS (PBS) y CISNE (CISNE); y los porcentajes de incremento del tiempo de ejecución calculados en base al tiempo de caracterización para PBS (%Inc PBS) y CISNE (%Inc CISNE). La fila final contiene los promedios de los tiempos de ejecución y los porcentajes.

El análisis de los resultados, tomando en cuenta información relacionada con el mapeo de las tareas por nodos, muestra que la gran ventaja de CISNE sobre PBS está en las políticas de distribución espacial, que considera la carga de los nodos a la hora de asignar alguna aplicación.

La Figura 6.2 muestra el tiempo de turnaround promedio, en negro, y el porcentaje de mejora, en rojo, de nuestro entorno con respecto a PBS. Para este experimento la carga carga paralela estaba compuesta de 20 benchmarks del NAS que arriban al sistema bajo una distribución de Poisson. Los resultados de CISNE son desglosados empleando las diferente políticas de asignación de cores existentes. Como puede observarse en la figura, el porcentaje de mejora para las políticas propuestas y LIN++ oscila entre el 18 % y el 21 %.

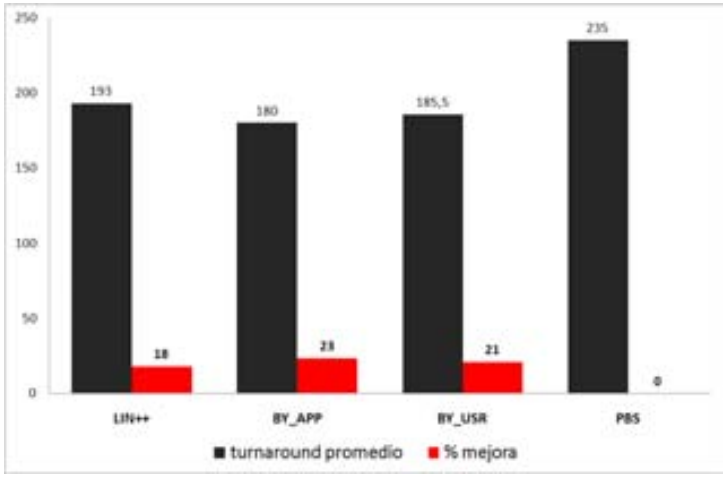

Figura 6.2: Tiempos de turnaround promedio y porcentaje de incremento en los tiempos de turnaround de PBS y CISNE, comparados contra los tiempos ideales de ejecución, desglosados por las diferentes políticas.

6.4. Rendimiento de las políticas de asignación de cores

En esta sección mostramos los resultados que avalan los objetivos para los que construimos nuestras políticas de asignación de cores.

6.4.1. Resultados para el entorno real

En esta sección mostraremos los resultados alcanzados por nuestras políticas en un entorno real, podemos encontrar la descripción de hardware y la configuración de software en 6.1.2.

6.4.1.1. Comparación entre las políticas propuestas

Para el experimento mostrado en la Figura 6.3, 8 aplicaciones paralelas arriban al sistema aleatoriamente. Entre estas 8 aplicaciones paralelas incluimos 2 de tipo SRT, que representan un 25 % del total de la carga de trabajo. Esta carga ha sido balanceada para que de las aplicaciones paralelas que se ejecutan a la vez consuman la CPU de forma diferente, dichas formas de uso de la CPU pueden ser CPU o IO bound. En el eje de las abscisas encontramos la información relacionada con la leyenda del contenido, la presencia o no de usuario local y el nombre de la política.

El análisis de la Figura 6.3 arroja los siguientes puntos:

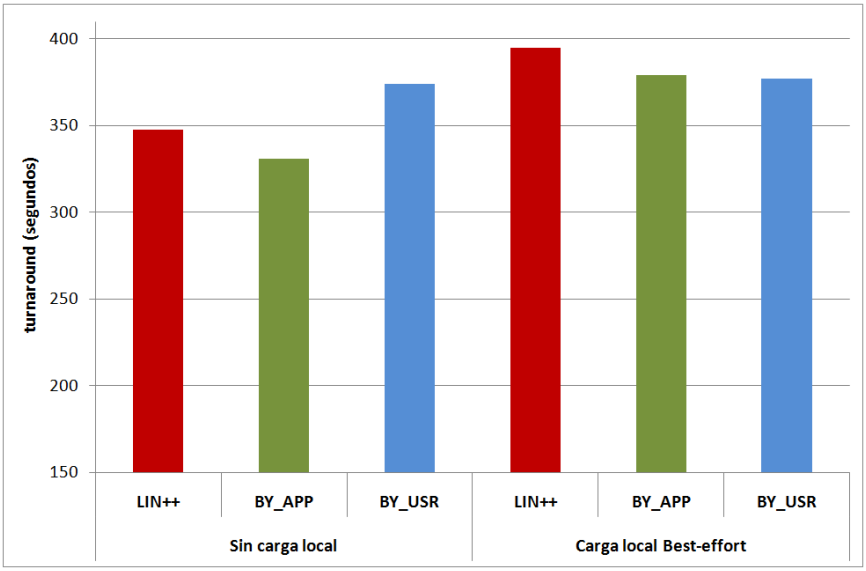

Figura 6.3: Tiempos de espera, ejecución y turnaround para las diferentes políticas y Linux, con y sin carga local. Carga de trabajo balanceada.

- Cuando no hay presencia de usuario local, la mejor política es la BY_APP, que mejora el comportamiento de LIN en un 4%, debido a que emplea los cores para los diferentes tipos de aplicaciones paralelas involucradas en el experimento. En cambio BY_USR obtiene un turnaround promedio un 8% peor que LIN , debido a que sobrecarga unos de los cores.

- Como es de esperar, la presencia de carga local afecta la ejecución de las aplicaciones paralelas, incrementando el turnaround promedio de las aplicaciones paralelas, aunque de forma diferente para cada política. En el caso de LIN el incremento es de un 12%, para BY_APP un 13% y para BY_USR un 1%. Las razones para estos comportamientos tan dispares están en el uso que hacen las diferentes políticas de los cores disponibles, y son las siguientes:

  - En el caso de BY_USR, la carga local local va a parar al core dedicado a ella, que antes estaba libre, por lo que apenas afecta el turnaround promedio de la carga paralela.
  - El caso de BY_APP, la carga local Best-effort va a afectar los tiempos de ejecución de los benchmarks paralelos tipo Best-effort, incrementando el turnaround promedio de la ejecución. Por otra parte, para LIN el efecto es similar al de BY_APP

- El comportamiento de los benchmarks remotos como SRT incluye una pausa aparte, el cumplimiento de los deadlines deseados. Para obtener

los resultados mostrados se realizaron 3 ejecuciones de cada caso, y se calculó su promedio. Para este experimento los deadlines siempre se cumplen para la política BY_APP, para LIN solo se cumple el 33% para el experimento sin carga local y para BY_USR un 33%, independientemente de la presencia o no de carga local.

Para mostrar la importancia del balanceo de carga de acuerdo al uso de la CPU en nuestro entorno hemos realizado un experimento en el cual la carga ha sido construida de tal forma que las aplicaciones paralelas arriban al sistema de forma explícitamente desbalanceada. Los resultados obtenidos son mostrados en la Figura 6.4. Esta configuración de carga de trabajo representa un desafío importante para nuestro entorno, ya que dificulta la aplicación de las políticas especiales para el balanceo de carga.

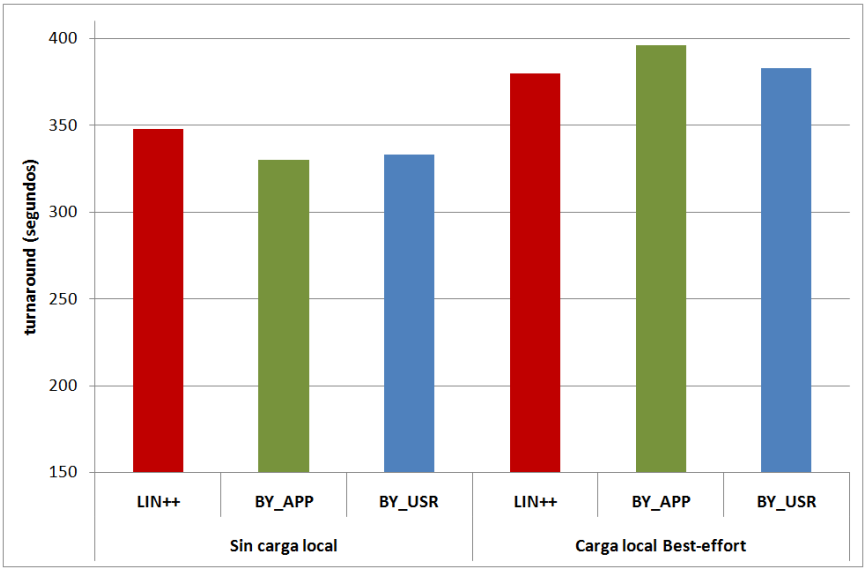

Figura 6.4: Tiempos de espera, ejecución y turnaround para las diferentes políticas y Linux, con y sin carga local. Carga de trabajo no balanceada.

Para el caso en que no hay carga local los resultados ambas políticas mejoran ligeramente a LIN , pero en presencia de carga local tenemos pérdidas del 4% para la política BY_APP y del 1% para BY_USR. Para este caso, con la carga paralela desbalanceada y en presencia de carga local, tenemos pérdidas de entre un 2% y un 4% para las políticas BY_USR y BY_APP con respecto a LIN .

6.4.2. Resultados para el entorno simulado

En esta sección presentamos los resultados obtenidos por nuestro simulador, empleando el núcleo de estimación simulado.

6.4.2.1. Comportamiento de las políticas

La Figura 6.5 muestra un experimento de simulación donde 8 aplicaciones paralelas arriban al entorno de planificación de forma aleatoria. La carga de trabajo que arriba al sistema incluye dos aplicaciones paralelas de tipo SRT. La carga local varía, representando los tres estados posibles para nuestro escenario: sin carga local y con carga local de tipo Best-effort o SRT. En el caso que se incluya carga local, está presente en todos los nodos, siendo los consumos de CPU y memoria por tipo de aplicación de 15% y 30% para la carga local Best-effort; y de 20% y 5.1% para las SRT.

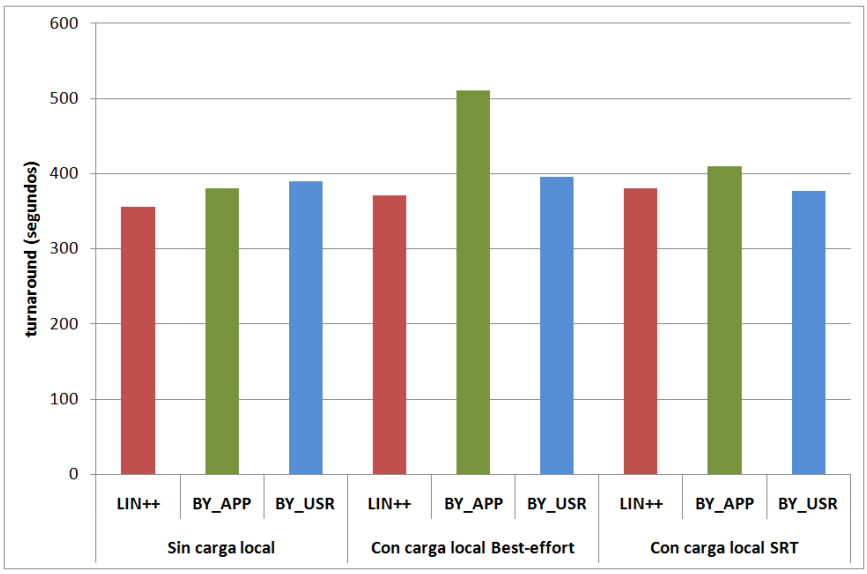

Figura 6.5: Comparación del rendimiento simulado (ST_SRT) entre Linux++ con y sin presencia de carga local Best-effort. Lanzadas 8 aplicaciones paralelas, de las cuales 2 son de tipo SRT.

En este experimento podemos apreciar la influencia de los diferentes tipos de carga local en el turnaround promedio de la carga para las diferentes políticas. El análisis de estos datos arroja los siguientes puntos:

- Política BY_APP: La predicción del turnaround promedio varía notablemente para los diferentes tipos de carga local:
    - Al incluir carga local Best-effort el turnaround promedio aumenta en un 2%, debido a un efecto que ya analizamos previamente en las ejecuciones reales, los benchmarks de tipo Best-effort se ven afectados por la carga local del mismo tipo, incrementando el turnaround promedio.

- Por otra parte, el turnaround promedio disminuye en casi un 20% al incluir carga local de tipo SRT, debido a los recursos que se liberan para el uso de las aplicaciones paralelas de tipo Best-effort. Hemos de tener en cuenta que las aplicaciones locales SRT no afectan el turnaround promedio debido a que si aceptamos una aplicación paralela SRT es porque estamos seguros de tener los recursos de CPU y memoria principal que necesita.

- Los resultados para la política BY_USR aparecen se ven afectados, debido a que los recursos para las aplicaciones locales de cualquier tipo se encuentran reservados.

- Es notable que nuestro simulador asigna una buena comportamiento a la política LIN , que tampoco se ve demasiado afectada por los diferentes tipos de carga local. Los incrementos son de un 4% al pasar de sin carga local a carga local Best-effort y de un 2% al pasar de carga local Best-effort a SRT. Recordemos que esta política considera el balance de carga paralela resultado de la planificación espacial de nuestro entorno.

Para este conjunto de experimentos todos los deadlines se cumplieron.

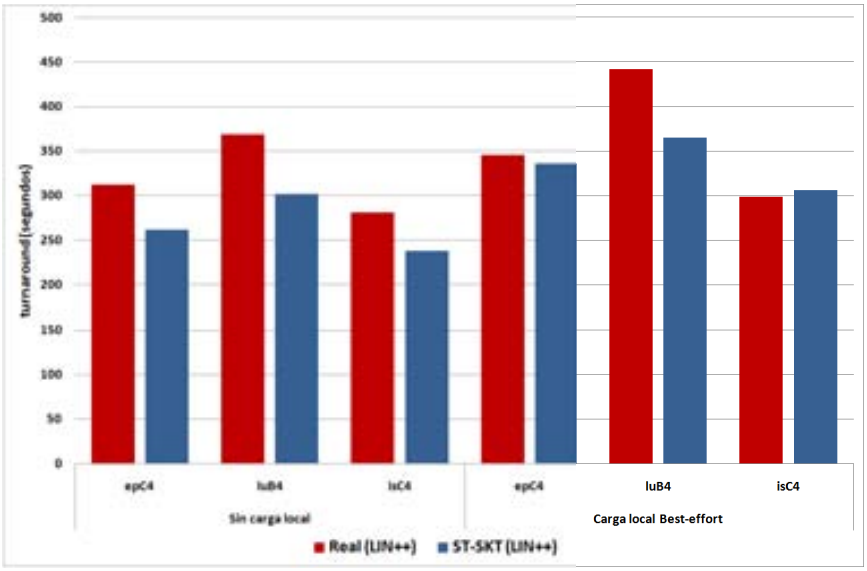

Figura 6.6: Comparación del método simulado (ST_SIKT) contra Linux con presencia de carga local Best-effort y sin ella.

6.4.2.2. Validación del simulador

Al contar con un entorno capaz de lanzar tanto ejecuciones reales como simuladas podemos validar nuestro simulador a través de la comparación de los resultados de ambos tipos de ejecuciones. Para la validación de los datos obtenidos mediante simulación los compararemos con casos reales, razón por la cual en esta sección sólo encontraremos experimentos que involucren 2, 4 u 8 nodos.

La Figura 6.6 muestra los resultados de un experimento de validación del simulador que implica tres clases diferentes de benchmarks NAS-MPI. La métrica empleada es turnaround y a través de las etiquetas en el eje de las abscisas podemos diferenciar si había usuario local o no, y el nombre del benchmark lanzado. La presencia de usuario local es en todos los nodos, y el tipo de aplicación es Best-effort. En este experimento no consideramos la presencia de aplicaciones paralelas de tipo SRT. Los resultados que arroja el simulador para este caso son de cerca del 7 % en el error promedio para en caso en el que no se considera carga local y del 16 % para cuando si se considera. También queremos destacar que la mayor desviación que encontramos no sobrepasa el 18 %.

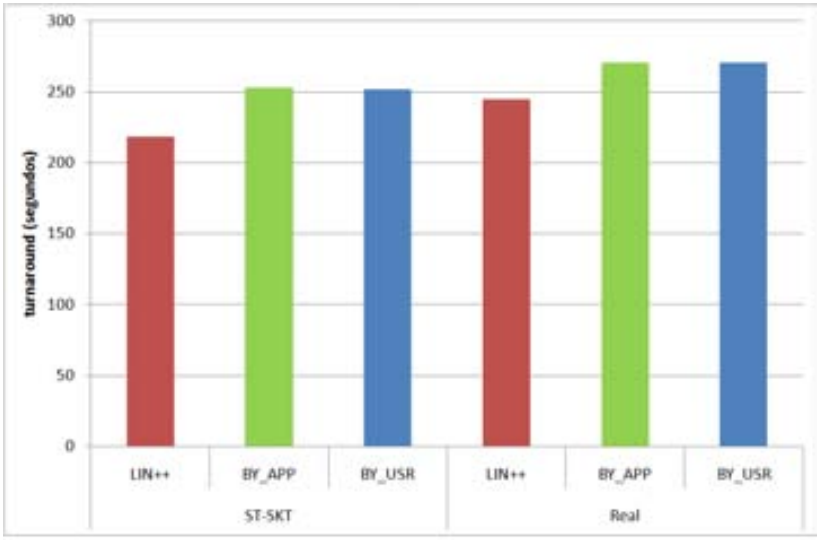

Figura 6.7: Comparación del método simulado (ST_SKT), desglosado para las diferentes políticas, contra ejecuciones reales. Experimento sin presencia de carga local.

Un elemento destacable del experimento mostrado en la Figura 6.6 es que nuestro esquema simulado reacciona correctamente a la presencia de carga local, aunque en el caso del benchmark isC4 notamos que la estimación es superior al valor de la ejecución real. Este comportamiento nos llama la

atención pues en todas las demás estimaciones el valor es menor. La causa de este comportamiento es la combinación de la generación de los eventos de comunicación en el simulador y la presencia de usuario local. Nuestro simulador considera que el usuario local de tipo Best-effort tiene un consumo del ancho de banda de red disponible, que combinado con el hecho de que el kernel IS es el que más cómputo genera, es la causa de esta situación.

Un experimento similar al anterior en lo referente a la carga, pero que involucra las diferentes políticas propuestas, es mostrado en la Figura 6.7. Para este caso, la carga está formada por tres benchmarks epC8, isC8 y luB8.srt, que arriban en el mismo instante de tiempo al sistema, uno de los cuales es de tipo SRT. Al igual que en el ejemplo anterior, los resultados del método simulado son más optimistas que la realidad, con el error en la predicción oscilando del 7 al 11 %. Para este experimento el deadline del benchmark, el luB8.srt, falló para LIN++ y BY_USR en el 66 % de los casos y se cumplió siempre para BY_APP.

6.5. La coplanificación en entornos multi-core

La coplanificación es una técnica de probada eficacia para mejorar los tiempos de turnaround de las aplicaciones paralelas en NOWs, en esta sección analizaremos los resultados obtenidos por nuestro entorno, para encontrar si obtenemos alguna mejora con los métodos que hemos empleado.

La Figura 6.8 muestra los resultados de lanzar una carga compuesta por las mismas aplicaciones paralelas, pero en diferente orden. El orden de arribo al sistema es un punto importante para nuestro entorno, pues nos permite aprovechar al máximo el potencial de cómputo existente combinando las cargas de acuerdo a si son CPU o IO bound a lo largo de los nodos del cluster. Una carga explícitamente desbalanceada, como la mostrada en la Figura 6.8 afecta directamente el rendimiento de nuestro entorno de planificación, pues impide que las políticas de planificación espacial diseñadas para sacar provecho del "bound" de las aplicaciones paralelas funcionen.

Los dos factores cuyo efecto mostramos con estos datos son:

1. Balanceo de la carga paralela en el cluster.
2. Longitud de las colas de tareas en los cores de las CPUs de los nodos.

Para analizar el efecto de la longitud de las colas en los cores hemos de comparar las diferencias entre los resultados con carga local y sin ella. La presencia de carga local, unido al MPL=4 empleado durante los experimentos, implica que cuando el experimento fue realizado con carga local cada procesador recibe dos tareas, que pueden ser ambas paralelas o una local

y una paralela. En cambio cuando no hay carga local, los cores reciben diferentes cantidades de tareas pertenecientes a aplicaciones paralelas. En la Figura 6.8 las barras rojas corresponden a los resultados con carga local, y las azules a los resultados obtenidos sin carga local. Como podemos observar, los resultados se diferencian para cada política, en dependencia de la finalidad de las mismas.

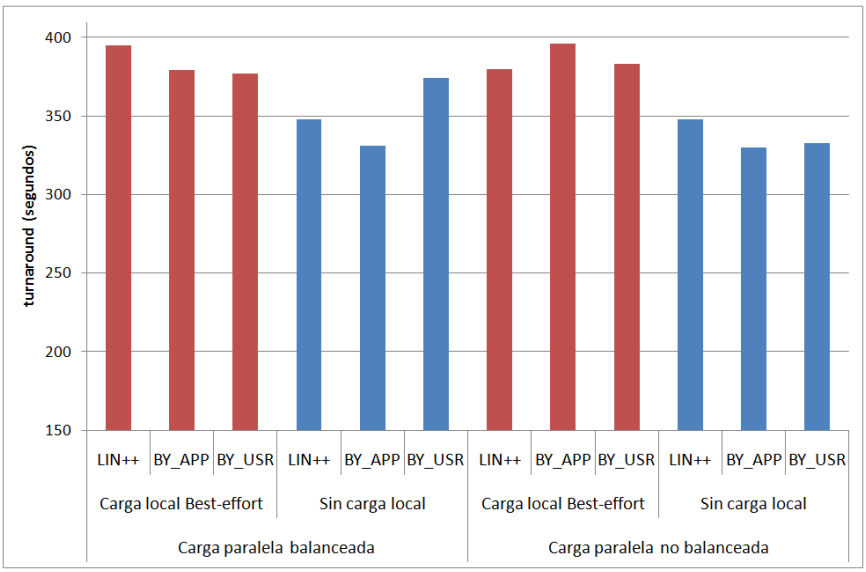

Figura 6.8: Efecto de la configuración de parámetros relacionados con la coplanificación sobre el turnaround de las aplicaciones paralelas.

La conjugación de los factores antes expuestos, balance de la carga paralela y longitud de las colas en los cores, afecta los resultados obtenidos. En el caso en que la carga paralela es balanceada y la longitud de las colas igual a 2 los resultados para ambas políticas serán buenos, comportándose de acuerdo a la finalidad para la que están definidas. También podemos observar que al no aplicar balance de carga el comportamiento de las políticas empeora, debido a la disminución de la probabilidad de ocurrencia de la coplanificación.

Capítulo 7

Conclusiones y Trabajo Futuro

En este capítulo enunciamos las conclusiones alcanzadas y las líneas de trabajo futuro.

7.1. Conclusiones

Nuestro trabajo es una contribución a la resolución del problema de la planificación de trabajos en NOWs no dedicadas . En este escenario existen múltiples tipos de aplicaciones locales y paralelas, algunas con características SRT; consideramos además que los procesadores de las estaciones de trabajo son multi-core. La unión de estas líneas de trabajo se hace necesaria para reflejar los cambios ocurridos en las aplicaciones a ejecutar en clusters no dedicados, con la complejidad adicional de un nuevo y más potente hardware. Incluimos nuevos tipos de aplicaciones locales SRT, cuyo mejor ejemplo son las aplicaciones multimedia, que implican una redefinición de las pautas que garantizan su coexistencia con la carga paralela. También tomamos en cuenta la evolución de las aplicaciones paralelas, que pueden requerir cierta QoS para una correcta ejecución. Afortunadamente conjuntamente a las crecientes demandas de recursos de las aplicaciones ha evolucionado el hardware, aunque con una tendencia que implica cambios importantes en las aplicaciones, la presencia de más de un core en los procesadores.

Creemos que las aulas de ordenadores presentes en cualquier universidad hoy en día son una fuente de poder de cómputo de la que muchos sistemas intentan hace uso eficiente. Un enfoque como el nuestro, que hace coexistir la carga paralela y la local implica un mejor uso de estos recursos. El cambio en las aplicaciones locales y paralelas anteriormente mencionado implica crear nuevos esquemas y métodos de planificación para que los usuarios locales no vean afectada la capacidad de respuestas de sus ordenadores. Si tomamos en

cuenta que las estaciones de trabajo ahora pueden tener procesadores multi-core, hemos de revisar si técnicas ya establecidas mantienen su vigencia, como es el caso de la coplanificación.

Como resultado del trabajo realizado para lograr los objetivos planteados se han desarrollado las modificaciones descritas en los capítulos 3 y 4, para la simulación y el entorno de ejecuciones reales respectivamente. Estas modificaciones son las encargadas de poner en práctica las políticas propuestas en el capítulo 2.

Sobre estas modificaciones nos gustaría destacar algunos puntos:

- El método de estimación por simulación es un simulador independiente, que podría ser extendido para realizar simulaciones sin necesidad de otros sistemas. Su implementación modular permite incorporar nuevos algoritmos de planificación con poco esfuerzo.
- Nuestro esquema de simulación a dos niveles ha probado ser una opción viable para las condiciones impuestas, con amplio potencial para probar tanto políticas de planificación temporal como espacial.
- El entorno para ejecuciones reales ha sido extendido para considerar procesadores multi-core y los nuevos tipos de aplicaciones.

Los resultados mostrados en el capítulo 6 apoyan nuestra tesis relacionada con el uso de la afinidad para generar políticas de asignación de cores. Las políticas creadas, BY_APP y BY_USR, están diseñadas para reforzar el comportamiento del sistema ante diferentes situaciones, obteniendo diferentes resultados de acuerdo al tipo de carga paralela y local en ejecución. En el caso de BY_USR, creada para proteger al usario local en un escenario más agresivo, alcanzamos resultados que demuestran que las reservas de recursos implementadas funcionan, al mostrar valores del turnaround promedio similares para los diferentes casos de presencia de usuario local. Para BY_APP, creada para mejorar el rendimiento del sistema para aplicaciones SRT, encontramos los mejores resultados en relación con la pérdida de deadlines de las aplicaciones paralelas SRT, del 100 % en los casos presentados.

Por otra parte, la comparación de la versión mejorada de CISNE contra PBS arroja buenos resultados, alcanzando mejoras en el turnaround promedio entre un 18 % y un 28 %, en dependencia del tipo de carga paralela. Cabe destacar que nuestro entorno brinda potencialidades relacionadas con las aplicaciones SRT que PBS aún no posee.

El uso de CISNE sin políticas de asignación de cores es factible, alcanzado mejores resultados que las políticas propuestas en dependencia de la situación. En cambio los resultados muestran que para situaciones donde se estimula la coplanificación las políticas basadas en asignación de cores

obtienen mejoras de entre el 4 % y el 7 %, con respecto a la política sin asignación de cores. Es válido destacar que las políticas se comportan de forma acorde con su objetivo.

El trabajo realizado a lo largo de esta investigación ha dado resultado a las siguientes publicaciones:

La etapa inicial de nuestra investigación involucró la creación del esquema de simulación a dos niveles, para el estudio de la planificación temporal de aplicaciones paralelas y locales, tanto Best-effort como SRT, aunque sin considerar procesadores multicore. También en esta etapa creamos los modelos analíticos capaces de representar nuestra problemática en ese momento. Estos resultados fueron publicados en los siguientes trabajos:

1. J. García, P. Hernández, J. Lérida, F.Giné, F.Solsona & M. Hanzich. ***Using Simulation for Job Scheduling with Best-Effort and Soft Real-Time Applications on NOWs.*** XVIII Jornadas de Paralelismo (CEDI 2007), Zaragoza, Spain, September 11 - September 14, 2007. pp. 415-422, ISBN:978-84-9732-593-6.

2. J. García, M. Hanzich, P. Hernández, E. Luque, J. Lérida, F. Giné, F. Solsona. ***Job Scheduling considering Best-Effort and Soft Real-Time***. XIII Congreso Argentino de Ciencias de la Computación (CACIC 2007), Corrientes y Resistencia, Argentina, October 1 - October 5, 2007. pp. 1239-1250, ISBN: 978-950-656-109-3.

Con el objetivo futuro de extender nuestro trabajo al multi-cluster, y como paso previo a la implantación de los resultados de este trabajo, llevamos a cabo los estudios mencionados más adelante. Creemos que ambos trabajos se complementan, creando un entorno de planificación para multi-clusters más flexible y potente.

1. Josep L. Lérida, Francesc Solsona, Francesc Giné, Mauricio Hanzich, Jose R. García & Porfidio Hernández. ***MetaLoRaS: A Re-scheduling and Prediction MetaScheduler for Non-dedicated Multiclusters.*** 14th European PVM/MPI User's Group Meeting, Paris, France, September 30 - October 3, 2007. LNCS, pp. 195-203, Vol: 4757/2007, ISBN: 978-3-540-75415-2.

2. Josep L. Lérida, Francesc Solsona, Francesc Giné, Jose R. García & Porfidio Hernández. ***Resource Matching in Non-dedicated Multicluster Environments.*** 8th International Conference, Toulouse, France, June 24 - June 27, 2008. LNCS, pp. 160-173, Vol: 5336/2008, ISBN: 978-3-540-92858-4.

Hemos propuesto nuevos métodos de predicción para clusters no dedicados en el trabajo.

1. Josep L. Lérida, Francesc Solsona, Francesc Giné, Jose R. García, Mauricio Hanzich & Porfidio Hernández. ***Enhancing Prediction on Non-dedicated Clusters.*** 14th International Euro-Par Conference, Las Palmas de Gran Canaria, Spain, August 26 - August 29, 2008. LNCS, pp. 233-242. Vol: 5168/2008, ISBN: 978-3-540-85450-0.

La validación de nuestro entorno de planificación, considerando tanto los nuevos tipos de aplicaciones como la presencia de procesadores multi-core y la propuesta del simulador a dos niveles, validada contra las ejecuciones reales, ha dado origen a las siguientes publicaciones:

1. Jose R. García, Josep L. Lérida & Porfidio Hernández. ***Resource manager with multi-core support for parallel desktop***. Proceedings of the 2009 IEEE International Conference on Cluster Computing, August 31 - September 4, 2009, New Orleans, Louisiana, USA. IEEE 2009, ISBN 978-1-4244-5012-1.
2. Jose R. García, Josep L. Lérida & Porfidio Hernández. ***Scheduling Soft Real-Time Applications on NOWs.*** 18th Euromicro Conference on Parallel, Distributed and Network-based Processing, Pisa, Italy, February 17 - February 19, 2010. pp. 59-63, ISBN: 978-0-7695-3939-3.

En estas publicaciones nos centramos en la planificación temporal de los nuevos tipos de aplicaciones soft real-time y su interacción con las aplicaciones Best-effort, considerando que los procesadores de las estaciones de trabajo sean multi-core. En los dos últimos trabajos hemos utilizado las políticas propuestas para la asignación de cores, BY_APP y BY_USR, con resultados positivos, al menos para una de ellas. Encontramos que la política BY_APP siempre obtiene un turnaround promedio mejor que Linux, y que empleando la política BY_USR los resultados varían de acuerdo a las condiciones del experimento. También comprobamos que ambas políticas obtienen mejor turnaround promedio que Linux cuando el cluster está ocupado por usuarios locales.

7.2. Trabajo Futuro

Nuestro trabajo se encuentra en un punto de inflexión importante, con varios frentes abiertos que ofrecen opciones muy interesantes, creemos que algunas de estas opciones para trabajo futuro son:

- La creación de políticas dinámicas de asignación de cores, que permitan balancear la longitud de las colas con heurísticas basadas en

las necesidades de planificación espacial y temporal, adaptándose a las necesidades de sus escenarios específicos.

- Las aplicaciones que se lanzan en nuestro entorno son un punto importante a considerar, sobre este tema en particular proponemos lo siguiente:

  - Considerar benchmarks o aplicaciones paralelas dependientes de los datos.
  - Refinar las caracterizaciones de las aplicaciones locales SRT, considerando puntos como obtener datos relacionados con los miss-deadlines y estadísticas relacionados con ellos.

- Nuestro esquema simulado puede ser extendido permite obtener estadísticas relacionadas con el comportamiento de la planificación temporal en cada nodo del cluster, con esta información disponible podrían crearse algoritmos de planificación tiempo real adaptados a nuestras necesidades.

- La planificación de aplicaciones en multi-clusters compite en importancia con otros enfoques relevantes como el Grid, con la ventaja de que en enfoques como el nuestro las estaciones de trabajo no tienen que estar marcadas como ociosas para su uso.

- En el momento actual, nuestro entorno solo lanza las tareas pertenecientes a una misma aplicación paralela en nodos diferentes, principalmente debido a los conflictos que generaría con nuestras políticas el colocarlos en el mismo nodo. También consideramos el hecho de que nuestro cluster de pruebas es dual-core, limitando esta opción a aplicaciones diseñadas para ejecutar con dos procesadores. Con una clara tendencia a aumentar la cantidad de cores por procesador, este enfoque deberá ser extendido en un futuro.

- Nuestro entorno para las aplicaciones reales asigna la prioridad y el quantum interno al planificador de forma estática, sin tener en cuenta que Linux, con su sistema de prioridades dinámica puede variar estos valores y afectar nuestra planificación. Es necesario relacionar ambos sistemas de prioridades, haciendo más dinámico el proceso, para mejorar los resultados del entorno.

- La planificación de trabajos se ha centrado principalmente en el manejo de recursos como la CPU y la memoria principal. No obstante, existe una gama importante de aplicaciones, dónde su ejecución en el sistema es intensiva en E/S. En esta situación las políticas tradicionales de planificación pueden causar problemas considerables de rendimiento.

Nuestro objetivo a corto plazo es incorporar a nuestro entorno aplicaciones bioinformáticas, y analizar sus requerimientos y necesidad de recursos; a efectos de poder ejecutarlas de forma eficiente.

Apéndice A

Uso del Entorno de Planificación

Las modificaciones introducidas al entorno CISNE implican cambios en la forma de uso del sistema, en este apéndice mostramos los elementos a tener en cuenta para lanzar aplicaciones considerando los cambios realizados.

Configuración del entorno

Los datos de inicialización del sistema han de estar accesibles a través de ficheros de configuración. En esta sección mencionamos los ficheros necesarios, sus ubicaciones y la información que han de contener. Los datos más importantes son los relacionados con la configuración de políticas empleadas por LoRaS para la planificación a largo plazo y la información de las políticas de asignación de cores a emplear por SRT_Scheduler y LoRaSd.

Subsist.	**Directorio**	**Descripción**
LoRaS	*$CISNE_ROOT/LoRaS/config*	Configuración del planificador a largo plazo
LoRaSd	*$CISNE_ROOT/LoRaSd/config*	Configuración del planificador temporal y el uso de las capacidades multicore

Tabla A.1: Ficheros de configuración del entorno.

Para el correcto funcionamiento del sistema ha de estar definida la variable de sistema *$CISNE_ROOT*, que ha de contener los ficheros de configuración y ejecutables de cada uno de los subsistemas involucrados en el entorno.

Una vez definida esta variable, hemos de proveer al sistema los ficheros de configuración por subsistema enumerados en la Tabla A.1.

Es importante destacar que la variable $CISNE_ROOT no solo define la información relacionada con los subsistemas. Esta variable también sirve como punto de entrada para los datos de caracterización de las aplicaciones del entorno, que se encuentran en la carpeta *$CISNE_ROOT/jobs*. Y en caso de emplearse el tipo de servicio stand también contiene los ficheros con la carga de trabajo, en el directorio *$CISNE_ROOT/wkls*.

Configuración del planificador espacial

Si la variable *$CISNE_ROOT* está definida, podemos iniciar el sistema con la siguiente línea de comandos:

$CISNE_ROOT/LoRaS/bin/LoRaS <fichero_configuracion>

donde el fichero de configuración *<fichero_configuracion>* ha de ser un fichero localizado en el directorio *$CISNE_ROOT/LoRaS/config* y contiene la configuración del sistema. En este fichero cada línea representa un valor de configuración de la siguiente forma:

$< tipo >< campo_1 >< campo_2 > \ ... < campo_n >$

Los tipos de registros que pueden aparecer y los valores asociados a cada son mostrados en la Tabla A.2.

Registro	***Campo***	***Descripción***
Servicio	*Tipo*	Tipo del registro con valor: *service*
	Service	Indica el modo de servicio del sistema, y dos de los valores posibles para este campo son: ***stand***: el sistema no recibe peticiones remotas de ejecución de aplicaciones. Lee la carga de aplicaciones a ejecutar desde un fichero (*$CISNE_ROOT/wkls/workload*). Empleado durante ejecuciones de *experimentación* o *desarrollo*. ***simulator***: no se configuran los sistemas *LoRaSd* remotos y se prepara el sistema para realizar una *simulación fuera de línea*.

Cola	*Tipo*	Tipo del registro con valor: *queue*.
	Politic	Identificador de la política que gestiona la cola. Este valor debe estar definido a partir del campo *Ident.* de un registro de política ya especificado.
	MAX-Procs T.Ej. Mem	Cantidad máxima de procesadores, tiempo de ejecución (segundos) o memoria (MB) que podrán solicitar los trabajos para ser incluidos en esta cola. Un valor de -1 se entenderá como *sin límite superior*.
	MIN Procs T.Ej. Mem	Cantidad mínima de procesadores, tiempo de ejecución (segundos) o memoria (MB) que deberán solicitar los trabajos para ser incluidos en esta cola. Un valor de -1 se entenderá como *sin límite inferior*.
	Ident.	Identificador de la cola.

Política	*Tipo*	Tipo del registro con valor: *politic*
	Ident.	Identificador dada a la política. Este valor se utiliza luego para asociar las colas a las políticas, por lo tanto ha de ser único.
	Política	Tipo de política definida. Este valor es utilizado por el sistema LoRaS para saber qué política debe generar para administrar las colas asociadas.
	Config.	Nombre de un fichero que contiene información sobre la configuración de la política. Este es un campo opcional.

Nodo	*Tipo*	Tipo del registro con valor: *node*.
	Nombre	Nombre del nodo (sin dominio).
	Nombre ext.	Nombre extendido que incluirá el dominio al que pertenece el nodo representado por el registro.
	IP	Dirección IP del nodo.
	Política	Política que administrará el nodo. Este campo deberá contener un valor previamente definido como *Identific.* en un registro de tipo política.

Tabla A.2: Valores de los registros de configuración del planificador espacial.

Un ejemplo de configuración del sistema podría ser de la siguiente forma:

service stand

politic politic1 FLEX2 conf_flex

queue politic1 -1 -1 -1 -1 -1 -1 FCFS queue1

node aopcso1 aopcso1.uab.es 192.168.65.231

.

.

node aopcso8 aopcso8.uab.es 192.168.65.238

Dónde estos registros y sus valores asociados tienen el siguiente significado:

- *service stand*: el sistema está configurado en modo de experimentación.
- *politic politic1 FLEX2 conf_flex*: política con identificación *politic1* de tipo Flexible (segunda versión: *FLEX2*) y cuya configuración se encuentra en el fichero *$CISNE_ROOT/LoRaS/policies/conf_flex*.
- *queue politic1 -1 -1 -1 -1 -1 -1 FCFS queue1*: cola con identificación *queue1,* gestionada por la política *politic1,* en la que se aceptarán todo tipo de trabajos (*-1 -1 -1 -1 -1 -1*) y que se encuentra ordenada a partir de una política de tipo *FCFS*.
- *node aopcsoX aopcsoX.uab.es 192.168.65.X politic1*: nodo *aopcsoX* gestionado por la política con identificación *politic1*.

Descripción de una carga

En el caso en que el sistema se utilice en modo de desarrollo o experimentación (*service stand* en la configuración general), entonces es necesario definir una *carga paralela*. Para tal definición se crea un nuevo fichero ubicado en *$CISNE_ROOT/wkls*, dentro del cuál se encuentran una secuencia arbitrariamente extensa de registros con formato <*tiempo_llegada*> <*trabajo*>, donde <*tiempo_llegada*> es el momento en que llegará la aplicación al sistema para ser planificada (medido en segundos) y <*trabajo*>, es el nombre de un fichero de descripción de trabajos que se ubica en *$CISNE_ROOT/jobs*.

Configuración del planificador temporal

A esta información necesaria para el planificador a largo plazo hemos de agregar los ficheros de configuración del planificador temporal. La información ha de ser colocada en el directorio

$CISNE_ROOT/LoRaSd/config

de dónde el demonio de LoRaSd la leerá de acuerdo a los parámetros que reciba.

En los ficheros que coloquemos en esta carpeta hemos de colocar la información en la forma:

$< identificador >< valor >$

para que pueda ser leída por el demonio de LoRaSd. Los valores que actualmente procesa nuestro entorno son los que encontramos en la Tabla A.3.

Identificador	**Valores posibles**	**Descripción**
policy	linux, by_app, by_user	Política a emplear para la asignación de las tareas a los cores, se corresponden con: linux=LIN++, by_app=BY_APP y by_user=BY_USR
affinity	true, false	Si se habilita o no la afinidad de cores, solo tiene sentido para la política linux
core_count	2	Cantidad de cores que tiene el procesador, en la actualidad solo se reconocen los procesadores dual-core

Tabla A.3: Valores de los registros de configuración del entorno.

Ejecución del entorno

En la carpeta *$CISNE_ROOT/scripts* podemos encontrar el script *runExperimentConfigLoRaSd,* que funciona recibiendo los siguientes parámetros de entrada:

- <*configuracion*>: el nombre de un fichero de configuración general del sistema ubicado en *$CISNE_ROOT/LoRaS/config/system.*
- <*politica*>: el nombre de un fichero de configuración para la política a emplear, ha de estar ubicado en la carpeta *$CISNE_ROOT/LoRaS/config/policies.*

- <*carga_paralela*>: nombre del fichero que contiene la carga paralela a ejecutar, ha de estar ubicado en
  $CISNE_ROOT/wkls.

- <*usuario*>: nombre de un usuario a emplear para lanzar el sistema y las aplicaciones paralelas, ha de contar con los permisos adecuados.

- <*política_asig_cores*>: fichero que contiene las especificaciones de la política de asignación de cores a emplear, ha de estar en la carpeta
  $CISNE_ROOT/LoRaSd/config.

- <*carga_local*>: indica si queremos utilizar carga local (1) o no (0).

- <*tipo_carga_local*>: el nombre de un fichero conteniendo la especificación del tipo de carga local deseada. Si no se especifica un camino completo, el fichero se busca en
  $CISNE_ROOT/scripts.
  Este fichero posee una línea por cada carga local que se desee, donde se especifica: <*nodo*> <*tipo*>, siendo <*nodo*> uno de los nodos utilizados por el sistema para la ejecución del experimento y <*tipo*> una identificación del perfil de usuario que se quiere para la carga local.

- <*salida*>: como salida del sistema se generan ficheros de estado, que reunen información sobre el consumo de recursos de las aplicaciones y los tiempos de inicio y fin de cada una. Se creará una carpeta en:
  $CISNE_ROOT/results/<*salida*>.
  con el nombre que hemos pasado como parámetro que contendrá los ficheros de resultados.

Apéndice B

Uso del Simulador

La configuración extra necesaria para emplear nuestra herramienta de simulación se debe principalmente a la necesidad de proveer al simulador de datos que no están accesibles desde el modo de simulación, como lo son el estado y características de los nodos o la carga local.

Configuración del simulador

Para lanzar el entorno en modo de simulación ha de realizarse toda la configuración mencionada previamente con algunos valores especiales en ciertos registros e introducir otros nuevos que contienen la información que ahora no se puede recolectar. La Tabla muestra los cambios y nuevos registros que hemos de incluir en nuestros ficheros de configuración para lanzar el entorno en modo simulación.

Registro	***Campo***	***Descripción***
Carga paralela	*Tipo*	Tipo del registro con valor: *workload*.
	Fichero	Nombre del fichero que contiene la carga de aplicaciones paralelas, cada línea ha de contener el tiempo en el que la aplicación paralela arriba al sistema y el fichero de caracterización de la misma.

Carga local	*Tipo*	Tipo del registro con valor: *local*.
	Nodo	*Identificador* del nodo de simulación afectado por la presencia de la carga local.
	Tiempo inicio	Tiempo de inicio de la carga local en el sistema.
	Tiempo fin	Tiempo de fin de la carga local en el sistema, para que la carga exista desde el *tiempo de inicio* hasta el fin de la simulación este campo debe tener el valor -1.
	Memoria	Porcentaje de memoria utilizada en el nodo.
	CPU	En caso de ser carga local Best-effort, este valor contiene el porcentaje de CPU utilizado por la carga local en el nodo. Si la carga local a simular es SRT este valor ha de ser igual a 0.
	deadline	En caso de ser carga local de tipo SRT, este valor representa el deadline de la tarea SRT. Si es de tipo Best-effort este valor ha de ser igual a 0.
	Período	En caso de ser carga local de tipo SRT, este valor representa el período de la tarea SRT. Si es de tipo Best-effort este valor ha de ser igual a 0.
	CpuBurst	En caso de ser carga local de tipo SRT, este valor representa la cantidad de CPU requerida periódicamente por la tarea SRT. Si es de tipo Best-effort este valor ha de ser igual a 0.

Nodo de simulación	*Tipo*	Tipo del registro con valor: *simnode*.
	Identific.	Identificador del nodo de simulación. Este valor ha de ser único.
	Mem. total	Cantidad total de memoria en el nodo.
	Mem. usada	Cantidad de memoria utilizada en el nodo al inicio del proceso de simulación.
	CPU	Porcentaje de CPU utilizada en el nodo al inicio del proceso de simulación.
	Potencia	Potencia de cómputo de CPU del nodo, medido en *bogomips*.

Tabla B.1: Principales parámetros de configuración del simulador.

Como en el Anexo A, mostramos un ejemplo de fichero de configuración del entorno para la simulación.

service simulator

simulate srtsim SRT sim_ output

politic politic1 FLEX2 conf_ flex

queue politic1 -1 -1 -1 -1 -1 -1 FCFS queue1

simnode node1 1024 100 25 6834 2 politic1

.

.

simnode nodeN 1024 100 25 6834 2 politic1

local node4 0 -1 50 15 0 0 0

local node5 0 -1 50 15 0 0 0

local node6 0 -1 50 0 0 200 25

local node7 0 -1 50 0 0 200 25

Esta configuración tiene el siguiente significado:

- *service, politic, queue*: se definen de la misma manera que para el funcionamiento normal del sistema, lo que nos permite reutilizar toda la configuración e implementación del gestor de colas y las políticas de planificación espacial.
- *simulate*: especifica el kernel de simulación a emplear, para emplear nuestro núcleo simulado la configuración ha de ser la mostrada en el ejemplo.
- *simnode*: cada uno de los registros *simnode* define un nodo utilizable por el sistema para el proceso de simulación. En el ejemplo se han definido 8 nodos con 1024 MB de memoria total, una potencia de CPU equivalente a 6834 bogomips y una carga inicial de memoria y CPU de 100MB y 25 % respectivamente.
- *local*: define un cargas locales para los nodos *node4, node5, node6* y *node7* que inician al principio del proceso de simulación (0) y se extienden hasta el final (-1) . En el caso del registro que especifica una carga local sobre el nodo *node4*, se define un 15 % (15) de consumo de CPU y una carga de memoria del 50 % (50) del total del nodo en cuestión. Por otra parte, la definición de *node6* incluye carga local de tipo SRT, al especificar el uso de memoria y ser los valores del período y cpuburst diferentes de 0.

- *workload*: define el fichero (*simworkload*) que contiene la carga de aplicaciones paralelas para la cual se desarrollará el proceso de simulación. El formato es igual al descrito para las ejecuciones reales, ha de contener un tiempo de arribo y fichero de caracterización de trabajo por línea, en el formato:
  $< tiempo\_de\_arribo >< fichero\_caract\_aplic\_paralela >$

Caracterización de los trabajos

Una cuestión de especial importancia es la definición las características de una aplicación para que el entorno pueda acceder a los valores que la caracterizan durante la simulación. Para proveer las descripciones de los trabajos, el entorno obtiene la información desde fichero definidos para cada trabajo, que posee los registros y campos especificados en la tabla B.2.

Registro	***Campos***	***Descripción***
Usuario	*Tipo*	Tipo del registro con valor: *user*.
	Nombre	Nombre del usuario del sistema.
Ejecutable	*Tipo*	Tipo del registro con valor: *exec*.
	Programa	Nombre del ejecutable de la aplicación paralela.
Directorio	*Tipo*	Tipo del registro con valor: *dir*.
	Directorio	Directorio donde se encuentra el ejecutable de la aplicación paralela.
Selección de trabajos	*Tipo*	Tipo del registro con valor: *params*.
	Valores	Valores de los parámetros que se han de pasar a la aplicación cuando sea ejecutada por el sistema.
Tipo de trabajo	*Tipo*	Tipo del registro con valor: *type*.
	Valor	Indica el tipo de aplicación para determinar el tipo de despachador que hemos de asociarle.
Conf. del tipo de trabajo	*Tipo*	Tipo del registro con valor: *typeconf*.
	Fichero	Fichero que puede contener datos de entrada o configuración propios de la aplicación.
Cant. mín. de procs.	*Tipo*	Tipo del registro con valor: *minproc*.

	Min. Procs.	Cantidad mínima de procesadores necesarias para la aplicación.
Cant. máx. de procs.	*Tipo*	Tipo del registro con valor: *maxproc*.
	Max. Procs.	Cantidad máxima de procesadores necesarias para la aplicación. En nuestro caso al tratar con aplicaciones *rígidas*, este valor coincide con el parámetro anterior.
Cant. mín. de mem.	*Tipo*	Tipo del registro con valor: *minmem*.
	Min. Mem.	Cantidad mínima de memoria necesaria para la aplicación.
Cant. máx. de mem.	*Tipo*	Tipo del registro con valor: *maxmem*.
	Max. Mem.	Cantidad máxima de memoria necesaria para la aplicación.
Cant. mín. de tiempo	*Tipo*	Tipo del registro con valor: *mintime*.
	Min. Tiempo.	Cantidad mínima de tiempo necesaria para la ejecución de la aplicación (en un entorno dedicado).
Cant. máx. de tiempo	*Tipo*	Tipo del registro con valor: *maxtime*.
	Max. Tiempo	Cantidad máxima de tiempo necesaria para la ejecución de la aplicación (en un entorno dedicado).
Cant. mín. de jiffies	*Tipo*	Tipo del registro con valor: *minjiffie*.
	Min. Jiffies.	Cantidad mínima de *jiffies* necesaria para la ejecución de la aplicación.
Cant. máx. de jiffies de aplicación	*Tipo*	Tipo del registro con valor: *maxjiffie*.
	Max. Jiffies	Cantidad máxima de *jiffies* que emplea la aplicación para realizar cómputo.

Cant. mín. de jiffies de aplicación	*Tipo*	Tipo del registro con valor: *appminjiff*.
	Min. Jiffies.	Cantidad mínima de *jiffies* que emplea la aplicación para realizar cómputo.
Cant. máx. de jiffies	*Tipo*	Tipo del registro con valor: *appmaxjiff*.
	Max. Jiffies	Cantidad máxima de *jiffies* necesaria para la ejecución de la aplicación.
Deadline máx.	*Tipo*	Tipo del registro con valor: *maxdeadline*.
	Deadline máximo	Tiempo máximo dentro del cual deberán concluir la aplicación paralela.
Orientac.	*Tipo*	Tipo del registro con valor: *bound*.
	Orientac.	Orientación de la aplicación: *CPU* o *IO* (entrada/salida).

Tabla B.2: Propiedades definidas para caracterizar un trabajo en CISNE.

Para que el entorno sea capaz de acceder a las descripciones de todos los trabajos caracterizados han de ser almacenados en: *$CISNE_ROOT/jobs*.

Ejecución del simulador

Como nuestro sistema reusa una mayor parte de los módulos del sistema real durante la simulación, podemos emplear el mismo sistema de *scripts* empleados para lanzar las ejecuciones reales. Las diferencias vienen dadas por los ficheros que recibe el script *runExperimentConfigLoRaSd*, explicado en la sección A. Este script ha de recibir una política definida de la forma descrita previamente en este anexo. Cabe recordar que la presencia de carga local para la simulación se hace a través del fichero de descripción del sistema.

Bibliografía

[1] ©Intel Corporation, Intel® Pentium® D Processor 950, http://processorfinder.intel.com/details.aspx?sSpec=SL95V.

[2] Xine: A free (gpl-licensed) high-performance, portable and reusable multimedia playback engine, http://www.xine-project.org/home.

[3] New capabilities help ease transition from traditional grid computing to cloud computing. http://sun.systemnews.com/articles/142/4/sw/22683, 12 2009.

[4] TF Abdelzaher, KG Shin, and N. Bhatti. User-level qos-adaptive resource management in server end-systems. *Computers, IEEE Transactions on*, 52(5):678–685, 2003.

[5] Luca Abeni and Giorgio C. Buttazzo. Integrating multimedia applications in hard real-time systems. In *RTSS*, pages 4–13, 1998.

[6] K. Aida. Effect of job size characteristics on job scheduling performance. *Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science*, 1911:1–17, 2000.

[7] DH Albonesi, R. Balasubramonian, SG Dropsbo, S. Dwarkadas, FG Friedman, MC Huang, V. Kursun, G. Magklis, ML Scott, G. Semeraro, et al. Dynamically tuning processor resources with adaptive processing. *Computer*, 36(12):49–58, 2003.

[8] J. Anderson and J. Calandrino. Parallel real-time task scheduling on multicore platforms. In *Proceedings of the 27th IEEE Real-Time Systems Symposium*, 2006.

[9] J. Anderson, J. Calandrino, and U. Devi. Real-time scheduling on multicore platforms. In *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2006.

[10] Bjorn Andersson, Sanjoy Baruah, and Jan Jonsson. Static-priority scheduling on multiprocessors. In *RTSS '01: Proceedings of the 22nd*

IEEE Real-Time Systems Symposium (RTSS'01), page 93, Washington, DC, USA, 2001. IEEE Computer Society.

[11] A. Andrzejak, P. Domingues, and L. Silva. Predicting machine availabilities in desktop pools. 2006.

[12] R.H. Arpaci, A.C. Dusseau, A.M. Vahdat, L.T. Liu, T.E. Anderson, and D.A. Patterson. The interaction of parallel and sequential workloads on a network of workstations. *ACM SIGMETRICS 1995*, pages 267–277, 1995.

[13] A. Arpaci-Dusseau. Implicit coscheduling: coordinated scheduling with implicit information in distributed systems. *ACM Transactions on Computer Systems*, 19(3):283–331, 2001.

[14] A. Atlas and A. Bestavros. Statistical rate monotonic scheduling. *Real-Time Systems Symposium, 1998. Proceedings., The 19th IEEE*, pages 123–132, 1998.

[15] A.K. Atlas and A. Bestavros. Slack stealing job admission control. Technical report, Technical Report BUCS-TR-98-009, Boston University, Computer Science Department, 1998.

[16] S. Bachthaler, F. Belli, A. Fedorova, and BC Burnaby. Desktop Workload Characterization for CMP/SMT and Implications for Operating System Design. In *Workshop on the Interaction between OS and Computer Architecture*, pages 2–9, 2007.

[17] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The nas parallel benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, 1991.

[18] S. Balle and D. Palermo. Enhancing an Open Source Resource Manager with Multi-Core/Multi-threaded Support. In *Job Scheduling Strategies for Parallel Processing*, pages 37–50. Springer, 2008.

[19] A. Bayucan, RL Henderson, JP Jones, C. Lesiak, B. Mann, B. Nitzberg, T. Proett, and J. Utley. Portable Batch System Administrator Guide, OpenPBS Release 2.3. *Veridian Information Solutions, Inc., Mountain View, CA, August*, 2000.

[20] E. Bini, G.C. Buttazzo, and G.M. Buttazzo. Rate monotonic analysis: the hyperbolic bound. *IEEE Transactions on Computers*, 52(7):933–942, 2003.

[21] B. Bode, D.M. Halstead, R. Kendall, Z. Lei, and D. Jackson. The portable batch scheduler and the maui scheduler on linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, 2000.

[22] GC Buttazzo and F. Sensini. Optimal deadline assignment for scheduling soft aperiodic tasks inhard real-time environments. *Computers, IEEE Transactions on*, 48(10):1035–1052, 1999.

[23] R. Buyya, M. Murshed, and D. Abramson. A deadline and budget constrained cost-time optimization algorithm for scheduling task farming applications on global grids. 2002.

[24] JM Calandrino, JH Anderson, and DP Baumberger. A hybrid real-time scheduling approach for large-scale multicore platforms. In *Real-Time Systems, 2007. ECRTS'07. 19th Euromicro Conference on*, pages 247–258, 2007.

[25] Stephen Childs and David Ingram. The linux-srt integrated multimedia operating system: Bringing qos to the desktop. *rtas*, 00:0135, 2001.

[26] Hao-Hua Chu and Klara Nahrstedt. A soft real time scheduling server in unix operating system. In *IDMS '97: Proceedings of the 4th International Workshop on Interactive Distributed Multimedia Systems and Telecommunication Services*, pages 153–162, London, UK, 1997. Springer-Verlag.

[27] M.L. Dertouzos. Control robotics: The procedural control of physical processes. *Information Processing*, 74:807–813, 1974.

[28] Desmoj Developer Team. Desmo-j project: http://asi-www.informatik.uni-hamburg.de/desmoj/.

[29] Advanced Micro Devices. Multi-core processors-the next evolution in computing, http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/Multi-Core_Processing_33211A.pdf, 2005.

[30] S.K. Dhall and CL Liu. On a real-time scheduling problem. *Operations Research*, 26(1):127–140, 1978.

[31] PA Dinda. A prediction-based real-time scheduling advisor. *Parallel and Distributed Processing Symposium., Proceedings International, IPDPS 2002, Abstracts and CD-ROM*, pages 10–17, 2002.

[32] P. Domingues, P. Marques, and L. Silva. Resource usage of windows computer laboratories. *International Conference on Parallel Processing Workshops (ICPPW'05)*, pages 469–476, 2005.

[33] B. Doytchinov, J. Lehoczky, and S. Shreve. Real-time queues in heavy traffic with earliest-deadline-first queue discipline. *Ann. Appl. Probab*, 11(2):332–378, 2001.

[34] Y. Etsion, D. Tsafrir, and D. G. Feitelson. Desktop scheduling: How can we know what the user wants? *ACM NOSSDAV*, pages 110–115, 2004.

[35] Yoav Etsion, Dan Tsafrir, and Dror G. Feitelson. Process prioritization using output production: Scheduling for multimedia. *ACM Trans. Multimedia Comput. Commun. Appl.*, 2(4):318–342, 2006.

[36] A. Fedorova, M. Seltzer, and M.D. Smith. Cache-fair thread scheduling for multicore processors. *Division of Engineering and Applied Sciences, Harvard University, Tech. Rep. TR-17-06*, 2006.

[37] Feitelson and Nitzberg. Job characteristics of a production parallel scientific workload on the NASA ames iPSC/860. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, volume 949, pages 337–360. Springer, 1995.

[38] D. Feitelson and L. Rudolph. Gang scheduling performance benefits for fine-grained synchronization. *Journal on Parallel and Distributed Computing (JPDC'92)*, 16(4):306–318, 1992.

[39] D. G. Feitelson, L. Rudolph, U. Schwiegelshohn, K. C. Sevcik, and P. Wong. Theory and practice in parallel job scheduling. *Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science*, 1291:1–34, 1997.

[40] E. Frachtenberg, D. G. Feitelson, J. Fernández, and F. Petrini. Parallel job scheduling under dynamic workloads. *Job Scheduling Strategies for Parallel Processing. High Perfomance Distributed Computing (HPDC'03), Seattle, Washington. Lecture Notes in Computer Science*, 2862.:208–227, June 2003.

[41] E. Frachtenberg, F. Petrini, S. Coll, and W. Feng. Gang scheduling with lightweight user-level communication. In *Proceedings of International Conference on Parallel Processing Workshops (ICPPW'01)*, Valencia, Spain, 2001. Workshop on Scheduling and Resource Management for Cluster Computing.

[42] Eitan Frachtenberg. Process scheduling for the parallel desktop. In *ISPAN '05: Proceedings of the 8th International Symposium on Parallel Architectures,Algorithms and Networks*, pages 132–139, Washington, DC, USA, 2005. IEEE Computer Society.

[43] D. Geer. Chip makers turn to multicore processors. *Computer*, 38(5):11–13, 2005.

[44] W. Gentzsch et al. Sun grid engine: Towards creating a compute power grid. In *Proceedings of the 1st International Symposium on Cluster Computing and the Grid*, page 35. Citeseer, 2001.

[45] F. Giné. *Cooperating Coscheduling: a coscheduling proposal for non-dedicated, multiprogrammed clusters*. PhD thesis, Universitat Autònoma de Barcelona, July 2004.

[46] K. Gopalan and K.D. Kang. Coordinated allocation and scheduling of multiple resources in real-time operating systems. In *Workshop on Operating Systems Platforms for Embedded Real-Time applications*, page 48. Citeseer.

[47] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel Computing*, 22(6):789–828, 1996.

[48] A. Gupta, B. Lin, and P. A. Dinda. Measuring and understanding user confort with resource borrowing. *13th IEEE International Symposium on high Performance and Distributed Compuing (HPDC'04), Honolulu, USA*, 2004.

[49] M. Hanzich. *A Temporal and Spatial Scheduling System for Non-dedicated Clusters*. PhD thesis, Universidad Autónoma of Barcelona, July 2006.

[50] M. Hanzich, F. Giné, P. Hernández, F. Solsona, and E. Luque. Coscheduling and multiprogramming level in a non-dedicated cluster. *EuroPVM/MPI 2004, Lecture Notes in Computer Science*, 3241:327–336, 2004.

[51] M. Hanzich, F. Giné, P. Hernández, F. Solsona, and E. Luque. 3DBackfilling: A space sharing approach for non-dedicated clusters. In *Parallel and Distributed Computing and Systems (PDCS'05)*, volume 17, pages 131–138. ACTA Press, 2005.

[52] M. Hanzich, F. Giné, P. Hernández, F. Solsona, and E. Luque. Cisne: A new integral approach for scheduling parallel applications on non-dedicated clusters. *EuroPar 2005, Lecture Notes in Computer Science*, 3648:220–230, 2005.

[53] M. Hanzich, F. Giné, P. Hernández, F. Solsona, and E. Luque. What to consider for applying backfilling on non-dedicated environments. *Journal on Computer Science and Technology*, 5(4):189–195, 2005.

[54] R. L. Henderson. Job scheduling under the portable batch system. *Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science*, 949:279–294, 1995.

[55] E. Hide, T. Stack, J. Regehr, and J. Lepreau. Dynamic cpu management for real-time, middleware-based systems. *Real-Time and Embedded Technology and Applications Symposium, 2004. Proceedings. RTAS 2004. 10th IEEE*, pages 286–295, 2004.

[56] C.H. Hsu and U. Kremer. The design, implementation, and evaluation of a compiler algorithm for CPU energy reduction. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, page 48. ACM, 2003.

[57] Intel - White Paper. Superior Performance with Dual-Core, ftp://download.intel.com/products/processor/xeon/srvrplatformbrief-.pdf, 2005.

[58] R. Jain, C. Hughes, and S. Adve. Soft real-time scheduling on simultaneous multithreaded processors. In *In 23rd IEEE International Real-Time Systems Symposium*, 2002.

[59] S. A. Jarvis, D. P. Spooner, H. N. Lim Choi Keung, J. Cao, S. Saini, and G. R. Nudd. Performance prediction and its use in parallel and distributed computing systems. *Future Generation Computer Systems, Special Issue on System Performance Analysis and Evaluation*, 2004.

[60] M. Jette and M. Grondona. Slurm: Simple linux utiltity for resource management. *ClusterWorld 2003 Conference and Expo*, June 2003.

[61] B. Kao and H. Garcia-Molina. Scheduling soft real-time jobs over dual non-real-time servers. *IEEE Transactions on Parallel and Distributed Systems*, 7(1):56–68, 1996.

[62] Evangelos Koukis and Nectarios Koziris. Memory bandwidth aware scheduling for smp cluster nodes. In *PDP '05: Proceedings of the 13th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP'05)*, pages 187–196, Washington, DC, USA, 2005. IEEE Computer Society.

[63] B. J. Lafreniere and A. C. Sodan. Scopred—scalable user-directed performance prediction using complexity modeling and historical data. *11th Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science*, 2005.

[64] T. Lechler and B. Page. Desmo-j: An object oriented discrete simulation framework in java. In *Proceedings of the European Simulation Symposium '99*, 1999.

[65] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: exact characterization andaverage case behavior. *Real Time Systems Symposium, 1989., Proceedings.*, pages 166–171, 1989.

[66] J. P. Lehoczky. Real-time queueing theory. In *RTSS '96: Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS '96)*, page 186, Washington, DC, USA, 1996. IEEE Computer Society.

[67] J. P. Lehoczky. Real-time queueing network theory. In *RTSS '97: Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS '97)*, page 58, Washington, DC, USA, 1997. IEEE Computer Society.

[68] Jochen Liedtke, Marcus Völp, and Kevin Elphinstone. Preliminary thoughts on memory-bus scheduling. In *EW 9: Proceedings of the 9th workshop on ACM SIGOPS European workshop*, pages 207–210, New York, NY, USA, 2000. ACM Press.

[69] C. Lin, H. Chu, and K. Nahrstedt. A soft real-time scheduling server on the windows nt. 1998.

[70] K.J. Lin and Y.C. Wang. The design and implementation of real-time schedulers in red-linux. *Proceedings of the IEEE*, 91(7):1114–1130, 2003.

[71] M. Litzkow, M. Livny, and M. Mutka. Condor- a hunter of idle workstations. *8th International Conference of Distributed Computing Systems*, 1988.

[72] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[73] W. Liu, V. Lo, K. Windisch, and B. Nitzberg. Non-contiguous processor allocation algorithms for distributed memory multicomputers. *IEEE/ACM Supercomputing 1994*, pages 227–236, November 1994.

[74] V. Lo, D. Zappala, D. Zhou, Y. Liu, and S. Zhao. Cluster computing on the fly: P2P scheduling of idle cycles in the internet. 2004.

[75] IBM LoadLeveler. User's Guide, Doc. No. Technical report, SH26-7226-00, IBM Corporation, 1993.

[76] Robert Love. Introducing the 2.6 kernel. *Linux J.*, 2003(109):2, 2003.

[77] S. Majumdar, D. L. Eager, and R. B. Bunt. Scheduling in multiprogrammed parallel systems. *ACM SIGMETRICS, Conference on Measurement and Modeling of Computer Systems*, pages 104–113, May 1988.

[78] S. Manolache, P. Eles, and Z. Peng. Task mapping and priority assignment for soft real-time applications under deadline miss ratio constraints. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(2):1–35, 2008.

[79] A. K. Mok. Fundamental design problems of distributed systems for the hard real-time environment. Technical report, Cambridge, MA, USA, 1983.

[80] A. W. Mu'alem and D. G. Feitelson. Utilization, predictability, workloads, and user runtime estimates in scheduling the ibm sp2 with backfilling. *IEEE Transaction on Parallel & Distributed Systems*, 12(6):529–543, 2001.

[81] M. Netto, R. N. Calheiros, R. K. S. Silva, C. F. De Rose, C. Northfleet, and W. Cirne. Transparent resource allocation to exploit idle cluster nodes in computational grids. In *Proceedings of The First IEEE International Conference on e-Science and Grid Computing*, pages 238–245, Melbourne, Australia, 2005. IEEE Computer Society.

[82] Novell. *Optimizing Linux for Dual-Core AMD Opteron Processors*. Novell, March 2006.

[83] J. Ousterhout. Scheduling techniques for concurrent systems. *Proceedings of the International Conference on Distributed Computing Systems*, 1982.

[84] F. Petrini and W. Feng. Improved resource utilization with buffered coscheduling. *Conference on High Performance Networking and Computing, Baltimore, Maryland. Proceedings of the 2002 ACM/IEEE conference on Supercomputing.*, 1-26, 2002.

[85] George & Sharma Akshay. Plale, Beth & Turner. Real time response to streaming data on linux clusters. Technical report, Indiana University. Computer Science Department Technical Report TR-569, November 2002.

[86] J. Regehr and J. Lepreau. The case for using middleware to manage diverse soft real-time schedulers. *Proceedings of the 2001 international workshop on Multimedia middleware*, pages 23–27, 2001.

[87] M. A. Rodriguez, F. Diaz del Río, C. Amaya, E. Florido, R. Senhadji, and G. Jiménez. Multicomputador hibernable: una solución para compartir los recursos de computación de los laboratorios docentes. *XIII Jornadas de Paralelismo, Lleida*, pages 117–122, 2002.

[88] L. Sha, T. Abdelzaher, K. Arzén, A. Cervin, T. Baker, A. Burns, G. Butazzo, M. Caccamo, J. Lehoczky, and A. K. Mok. Real time scheduling theory: A historical perspective. *Real-Time Systems*, 28:101–155, 2004.

[89] L. Sha, J.P. Lehoczky, and R. Rajkumar. Solutions for some practical problems in prioritized preemptive scheduling. *IEEE Real-Time Systems Symposium*, pages 181–191, 1986.

[90] J. Shirako, N. Oshiyama, Y. Wada, H. Shikano, K. Kimura, and H. Kasahara. Compiler control power saving scheme for multi core processors. *Languages and Compilers for Parallel Computing*, pages 362–376.

[91] Allan Snavely and Dean M. Tullsen. Symbiotic jobscheduling for a simultaneous mutlithreading processor. *SIGPLAN Not.*, 35(11):234–244, 2000.

[92] Q. O. Snell, M. J. Clement, and D. B. Jackson. Preemption based backfill. *Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science*, 2537:24–37, July 2002.

[93] P. Sobalvarro, S. Pakin, W. Weihl, and A. Chien. Dynamic coscheduling on workstation clusters. *Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science*, 1459:231–256, 1998.

[94] P. Sobalvarro and W. Weihl. Demand-based coscheduling of parallel jobs on multiprogrammed multiprocessors. *Job Scheduling Strategies for Parallel Processing, Lecture Notes in Computer Science*, 949:106–126, 1995.

[95] F. Solsona. *Coscheduling Techniques for Non-Dedicated Cluster Computing.* PhD thesis, Universitat Autònoma de Barcelona, 2002.

[96] F. Solsona, F.Giné, P. Hernández, and E. Luque. Implementing explicit and implicit coscheduling in a pvm environment. *EuroPar 2000, Lecture Notes in Computer Science*, 1900:1165–1170, 2000.

[97] F. Solsona, F. Giné, P. Hernández, and E. Luque. Predictive coscheduling implementation in a non-dedicated linux cluster. *EuroPar 2001, Lecture Notes in Computer Science*, 2150:732–741, 2001.

[98] Francesc Solsona, Francesc Giné, Josep L. Lérida, Porfidio Hernández, and Emilio Luque. Monito: A communication monitoring tool for a pvm-linux environment. In Jack Dongarra, Péter Kacsuk, and Norbert Podhorszki, editors, *PVM/MPI*, volume 1908 of *Lecture Notes in Computer Science*, pages 233–241. Springer, 2000.

[99] M. Spuri and GC Buttazzo. Efficient aperiodic service under earliest deadline scheduling. *Real-Time Systems Symposium, 1994., Proceedings.*, pages 2–11, 1994.

[100] Anand Srinivasan and James H. Anderson. Optimal rate-based scheduling on multiprocessors. *J. Comput. Syst. Sci.*, 72(6):1094–1117, 2006.

[101] S. Srinivasan, R. Kettimuthu, V. Subrarnani, and P. Sadayappan. Characterization of backfilling strategies for parallel job scheduling. *International Conference on Parallel Processing Workshops (ICPP-PW'02)*, pages 514–522, 2002.

[102] S. Srinivasan, V. Subramani, R. Kettimuthu, P. Holenarsipur, and P. Sadayappan. Selective buddy allocation for scheduling parallel jobs on clusters. In *Proceedings of IEEE International Conference on Cluster Computing (CLUSTER 2002)*, September 2002.

[103] H. Streich and M. Gergeleit. On the design of a dynamic distributed real-time environment. In *WPDRTS '97: Proceedings of the 1997 Joint Workshop on Parallel and Distributed Real-Time Systems (WPDRTS / OORTS '97)*, page 251, Washington, DC, USA, 1997. IEEE Computer Society.

[104] Jay K. Strosnider, John P. Lehoczky, and Lui Sha. The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. *IEEE Trans. Comput.*, 44(1):73–91, 1995.

[105] D. Talby and D. G. Feitelson. Improving and stabilizing parallel computer performance using adaptive backfilling. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05)*, page 84.1, Washington, DC, USA, 2005. IEEE Computer Society.

[106] S.R. Thuel and J.P. Lehoczky. Algorithms for scheduling hard aperiodic tasks in fixed-priority systems using slack stealing. *Real-Time Systems Symposium*, pages 22–33, 12 1994.

[107] D. Tsafrir, Y. Etsion, and D. G. Feitelson. Backfilling using runtime predictions rather than user estimates. Technical Report TR 2005-5, School of Computer Science and Engineering, Hebrew University of Jerusalem, November 2005.

[108] S. Venkataramaiah and J. Subhlok. Performance prediction for simple cpu and network sharing. *LACSI Symposium 2002*, 2002.

[109] Q. Wu, P. Juang, M. Martonosi, and D.W. Clark. Formal online methods for voltage/frequency control in multiple clock domain microprocessors. *ACM SIGPLAN Notices*, 39(11):248–259, 2004.

[110] M.T. Yang, R. Kasturi, and A. Sivasubramaniam. An automatic scheduler for real-time vision applications. *Parallel and Distributed Processing Symposium., Proceedings 15th International*, page 8, 2001.

[111] X. Zeng and A. Sodan. Job scheduling with lookahead group matchmaking for time/space sharing on multi-core parallel machines. In *Job Scheduling Strategies for Parallel Processing: 14th International Workshop, Jsspp 2009, Rome, Italy, May 29, 2009, Revised Papers*, page 232. Springer, 2009.

[112] Y. Zhan and A. Sivasubramaniam. Scheduling best-effort and real-time pipelined applications on time-shared clusters. *Proceedings of the 13th Annual ACM symposium on Parallel Algorithms and Architectures (SPAA'2001)*, pages 209–218, 2001.

[113] J. Zhang, T. Hamalainen, and J. Joutsensalo. A new mechanism for supporting differentiated services in cluster-based network servers. *mascots*, 00:0427, 2002.

[114] Y. Zhang, H. Franke, J. E. Moreira, and A. Sivasubramaniam. A comparative analysis of space- and time-sharing techniques for parallel job scheduling in large scale parallel systems. *IEEE Transactions on Parallel and Distributed Systems*, 2002.

[115] Y. Zhang, H. Franke, J. E. Moreira, and A. Sivasubramaniam. An integrated approach to parallel scheduling using gang-scheduling, backfilling, and migration. *IEEE Transactions on Parallel and Distributed Systems*, 14(3):236–247, March 2003.

[116] W Zhu. Allocating soft real-time tasks on clusters. *SIMULATION77*, 5-6:219–229, 2001.