Capítulo 4

Implementación de CSC en un Entorno LINUX-PVM

En este capítulo se describe la implementación del CoScheduling Cooperativo (*CSC)*, descrito en el capítulo anterior, en un entorno *PVM-Linux*. La descripción de la la implementación de CSC se inicia con una explicación detallada de todos aquellos conceptos, sobre Linux y PVM, necesarios para poder comprender la implementación realizada. Posteriormente, la implementación de cada uno de los módulos que configuran CSC es descrita en detalle.

La implementación realizada de CSC muestra como un s.o. de propósito general, como es el caso del s.o. Linux, puede ser adaptado, con unas pocas modificaciones de su código y junto con el apoyo de una librería de paso de mensajes, como es el caso de PVM, a las necesidades tanto de las aplicaciones locales, como de las distribuidas que se ejecutan a lo largo de un cluster no dedicado.

4.1. Entorno PVM-Linux: Conceptos Preliminares

CSC ha sido implementado en parte en el espacio del usuario y más concretamente, en el propio daemon de PVM, así como en el propio s.o Linux. Con objeto de facilitar la posterior descripción de la implementación realizada, en esta sección se describen las principales partes del kernel de Linux (v.2.2.15) que se han utilizado, y en algunos casos modificado, para la implementación de CSC. Asimismo, se describe el funcionamiento del entorno de paso de mensajes PVM, junto con las principales estructuras que se han utilizado para la implementación de CSC.

4.1.1. El descriptor de procesos de Linux

Linux necesita mantener información acerca del estado de cada uno de los procesos que gestiona. Con esta finalidad, el kernel asocia a cada proceso una estructura, denominada *task_struct*, cuyos campos contienen toda la información del proceso asociado. Los campos más relevantes de *task_struct* que se han utilizado a lo largo de la implementación de CSC son los siguientes:

- *uid*. Identificador del usuario propietario del proceso.
- *pid*. Identificador de proceso.
- *state*. Estado de un proceso Linux.
- *rt_priority*. Prioridad estática de los procesos de tiempo real. Los procesos en tiempo real pueden tener una prioridad estática comprendida entre 1 y 99, mientras que para los procesos de tiempo compartido este campo es igual a 0.
- *policy*. Política de planificación de Linux. Linux, siguiendo las especificaciones POSIX.1b [BC01], ofrece tres tipos diferentes de políticas de planificación, una para procesos de tiempo compartido (*SCHED_OTHER*), con una prioridad estática 0, y dos más para aplicaciones de tiempo real, (*SCHED_FIFO*) y (*SCHED_RR*), con una prioridad estática entre 1 y 99, de manera que para cada nivel de prioridad existe una cola gestionada de acuerdo con la política aplicada. Solamente procesos con privilegios de *superusuario (root)* podrán ejecutarse en los modos SCHED_FIFO y SCHED_RR. El planificador de Linux busca la cola no vacía con una prioridad estática superior y captura el proceso cabecera de la cola. La política de scheduling determina como se moverá cada proceso dentro de su cola, de acuerdo con los siguientes criterios:
  - SCHED_FIFO. Cada proceso planificado bajo esta política posee el control de la CPU, siempre y cuando no exista un proceso con una prioridad superior, el proceso se bloquee a si mismo mediante una llamada a *sched_yield()*, o bien se produzca una interrupción. Aquel proceso que sea adelantado por otro más proritario, será puesto a la cabecera de su cola de prioridad.
  - SCHED_RR. La única diferencia respecto de la política anterior, es que cada proceso se ejecuta durante un quantum de tiempo, de manera que una vez superado este quantum, el proceso será almacenado al final de su cola correspondiente.

- SCHED_OTHER. Este es el planificador típico de UNIX de tiempo compartido basado en una prioridad dinámica (*priority*), determinada por el "*nice*" valor, en un rango comprendido entre 0 (más prioritario) a 40 (menos prioritario), aunque los valores menores de 21 solamente pueden ser asignados por el superusuario. Un proceso, en este modo, solamente está permitido ejecutarse durante una cantidad de tiempo prefijada por el valor de su correspondiente contador (*counter*), el cual es inicializado con su correspondiente prioridad (*priority*), al inicio de cada época del planificador, y decrementado cada tick[1] del reloj.

- *counter*. Este campo indica el número de ticks de reloj que le quedan al proceso antes de que su quantum de tiempo expire.

- *priority* (prioridad dinámica para los procesos de tiempo compartido). Este es el quantum base asociado a cada proceso de tiempo compartido. Por defecto, este valor vale DEF_PRIORITY[2].

- *start_time*: Este campo contiene una referencia temporal respecto al inicio de la ejecución del proceso asociado, contada como el número de *ticks* transcurridos desde al arranque del sistema.

- *files*. Este campo apunta a la estructura *files_struct*, la cual especifica que ficheros están abiertos por el proceso.

- *maj_flt*. Número de fallos de página con acceso a disco realizados por dicho proceso.

- *nswap*. Número de páginas del proceso guardadas en el fichero de intercambio (*swap*).

- *next_task, prev_task*. Con el objetivo de incrementar la eficiencia en la búsqueda de un proceso de un determinado tipo (p.e. procesos preparados para la ejecución), el kernel ordena los procesos en diferentes listas circulares doblemente enlazadas. Los campos *prev_task* y *next_task* de cada *task_struct* son utilizados para implementar dichas listas, tal como se muestra en la figura 4.1. La cabecera de la lista es el proceso *init_task*, de manera que el campo *prev_task* de *init_task* apuntará al último proceso insertado en la lista.

[1] 1 tick=10ms
[2] DEF_PRIORITY=21 ticks=210ms

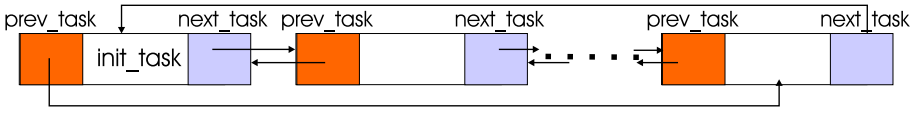

Figura 4.1: Lista de procesos.

- *mm*. Este campo apunta a la estructura *mm_struct*, la cual contiene toda la información relacionada con el espacio de direcciones de un proceso. Los campos de dicha estructura utilizados a lo largo de la implementación de *CSC* son los siguientes:

  - *rss*. Número de páginas del proceso residentes en memoria.
  - *total_vm*. Número total de páginas que ocupa el espacio de memoria virtual de dicho proceso.

4.1.2. El planificador de Linux

El planificador de Linux, implementado en la función *schedule()* del kernel, es llamado en los siguientes casos:

- Si el proceso en curso ha finalizado su ejecución.
- Si ha finalizado el quantum de tiempo asignado al proceso actualmente planificado.
- En caso de que el proceso en curso se encuentre bloqueado, esperando la finalización de un determinado evento.
- Al finalizar una llamada a sistema, es decir, justo antes de que el proceso pase de modo sistema a modo usuario.

Cada vez que el planificador de LINUX es ejecutado, se realizan los siguientes pasos de acuerdo con el orden enumerado a continuación:

1. **Gestión de la cola de interrupciones pendientes**. Existen diferentes interrupciones cuyo procesamiento puede ser retardado por el kernel. Con este fin, Linux dispone de unas rutinas específicas para el tratamiento de estas interrupciones, denominadas *Bottom Half Handlers* (*BH*), asociadas a diferentes periféricos. Como se puede ver en la figura 4.2, cada *BH* tiene asociado un flag (*Bh_active*) que es activado, en caso de que reste pendiente el procesamiento de la correspondiente

interrupción. El planificador, al inicio de su ejecución, comprueba dichos flags y en caso de que algún flag esté activado, procederá a procesar la correspondiente rutina de servicio de interrupción. En concreto, los bottom half utilizados en la implementación de CSC son los asociados con el teclado y el ratón, los cuales se denominan *KEYBOARD_BH* y *MOUSE_BH*, respectivamente. La consulta de los flags asociados con el teclado y el ratón permitirán conocer si hay algún usuario interactivo trabajando en dicha máquina.

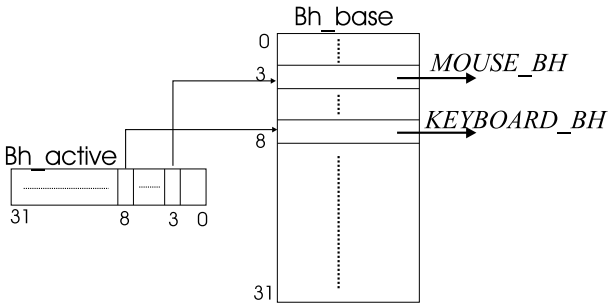

Figura 4.2: Bottom Half handlers.

2. **Proceso actual**. El proceso actual debe ser procesado, antes de que cualquier otro pueda ser seleccionado, siguiendo las siguientes pautas:

   - Si la política de planificación es *SCHED_RR*, el proceso es puesto al final de la cola de ejecución.
   - Si el proceso está interrumpido, es decir su estado es *TASK_INTERRUPTIBLE*, y ha recibido alguna señal desde la última vez que se ejecutó el planificador, su estado cambia a *TASK_RUNNING*.
   - En caso de que el estado de la tarea actual sea *TASK_RUNNING*, éste se mantiene.
   - En caso de que no se cumpla ninguna de las condiciones anteriores se borra de la cola de preparados (Ready_Queue).

3. **Selección de proceso**. Se selecciona el proceso de la cola de preparados más prioritario. El algoritmo 1 muestra el algoritmo utilizado. Este algoritmo recorre toda la cola de preparados y selecciona aquel proceso que tiene una mayor prioridad (*next*). La función *goodness* retorna la prioridad de planificación de cada proceso, de acuerdo con las políticas de planificación explicadas anteriormente. En la sección 4.2.2 se

describe en detalle el algoritmo implementado por la función *goodness* para calcular dicho valor.

Algoritmo 1 Selección del proceso a ejecutar.

```
1  c := -1000;
2  p := init_task->next_run;
3  while(p != init_task){
4    weight := goodness(p);
5    if(weight>c){
6      c := weight;
7      next := p;
8    }
9    p :=p->next_run;
10 }
```

4. **Cambio de contexto**. En caso de que el proceso seleccionado no sea el proceso actual, se tendrá que realizar el correspondiente cambio de contexto, lo que supone actualizar los campos correspondientes de la estructura *task_struct* asociada al proceso reemplazado.

5. **Reasignación de quantum**. El planificador de Linux divide el tiempo en *épocas*, de manera que una *época* finaliza cuando todos los procesos de la cola de preparados han consumido su quantum de tiempo, es decir, cuando todos ellos tienen el campo *counter* igual a cero. En este caso, el planificador procederá a resignar un nuevo quantum a todos los procesos residentes en dicha máquina, de acuerdo con el algoritmo 2.

Algoritmo 2 Reasignación del quantum.

```
1  for_each_task(p)
2    p->counter := (p->counter >> 1) + p->priority;
```

Este algoritmo tiende a combinar dos factores: el historial del proceso y la prioridad del proceso. Después de aplicarse el algoritmo, un proceso retendrá la mitad del *counter* que le había quedado después de la última operación de renovación de quantum, con lo que se conserva un antecedente del comportamiento reciente del proceso. Los procesos que se ejecutan todo el tiempo tienden a agotar su quantum rápidamente, pero, en cambio, los que pasan una buena parte de su tiempo

suspendidos pueden acumular ticks de planificación a lo largo de varias renovaciones y, por consiguiente, después de una renovación, terminarán con un mayor quantum (*counter*). Por lo tanto, el planificador de Linux favorece a aquellos procesos con altas tasas de E/S. Sin embargo, cabe destacar que el valor de *counter* nunca llegará a ser el doble que el valor de *priority*[3]. Una vez realizado el proceso de reasignación, descrito en el algoritmo 2, el planificador retornará al punto 3, con el fin de seleccionar un nuevo proceso para ejecución.

4.1.3. Subsistema de memoria del s.o. Linux

La gestión de memoria en Linux tiene dos componentes: el *sistema de gestión de memoria física,* que se encarga de asignar y liberar páginas, grupos de páginas y bloques pequeños de memoria y el *sistema de gestión de la memoria virtual.* En esta sección se describirán ambos componentes, haciendo especial énfasis en aquellas partes que interactuan directamente con el bloque de gestión de memoria de CSC.

4.1.3.1. Gestión de memoria física

El administrador primario de memoria física del núcleo de Linux es el asignador de páginas, el cual se encarga de asignar y liberar todas las páginas físicas, así como asignar, en caso de que se le solicite, intervalos de páginas contiguas físicamente. El asignador usa el *algoritmo de Buddy* para seguir el rastro de las páginas disponibles. El algoritmo de Buddy junta unidades adyacentes de memoria asignable. Cada región de memoria asignable tiene una compañera, y siempre que dos regiones contiguas quedan libres, se combinan para formar una región más grande. Esta región mayor también puede tener una compañera con la que puede combinarse para formar una región todavía mayor. Como alternativa, si una solicitud de memoria pequeña no puede satisfacerse asignando una región libre del tamaño requerido, una región libre mayor se subdividirá en dos compañeras para satisfacer la solicitud. La figura 4.3(a) muestra un ejemplo de asignación de un bloque de 4KB, en un sistema donde la región libre más pequeña es de 16KB, de manera que la región se divide recursivamente hasta que se tiene el tamaño deseado. Linux gestiona estos bloque de páginas libres por medio de un vector denominado *free_area.* Cada posición *iésima* del vector apunta a una lista que contiene todos aquellos de bloques de orden 2^i páginas libres, de manera que el bloque

[3]Asumiendo *counter* y *priority* igual a P, la serie geométrica Px(1+1/2+1/4+1/8+...) converge a 2xP.

más pequeño es de una página y el bloque más grande es de 512 páginas (ver figura 4.3(b)).

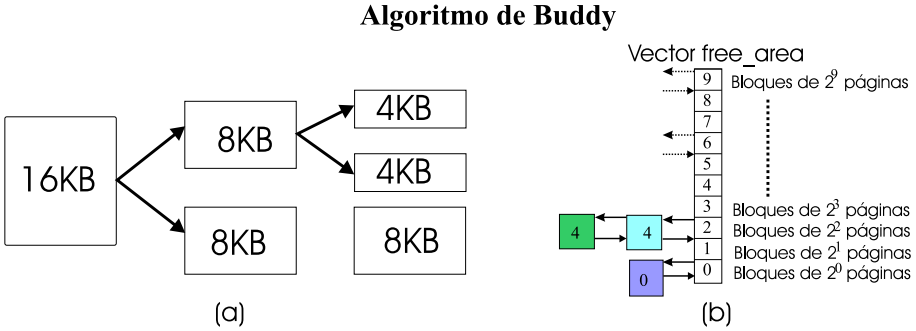

Figura 4.3: Algoritmo de Buddy.

En última instancia, todas las asignaciones de memoria en el núcleo de Linux ocurren estáticamente, mediante controladores que reservan una área contigua de la memoria durante el arranque del sistema, o bien dinámicamente, por medio del asignador de páginas. No obstante, las funciones del núcleo no tienen por que usar el asignador básico para reservar memoria; existen varios subsistemas de gestión de memoria especializados que usan el asignador de páginas subyacentes para administrar su propia reserva de memoria. Los subsistemas más importantes son el asignador de longitud variable *kmalloc* y las tres caches de disco del núcleo (cache de buffers, de páginas y de swap). Una cache de disco es un mecanismo software que permite al sistema guardar en la memoria principal datos que provienen de disco, con objeto de reducir los accesos y minimizar los tiempos de transferencia de datos.

La *cache de buffers* es la cache principal del núcleo para dispositivos orientados a bloques, como las unidades de disco, y es el mecanismo principal para efectuar una E/S con esos dispositivos. La *cache de páginas* almacena páginas enteras del contenido de archivos y no está limitada a los dispositivos de bloques. También puede almacenar datos de red y lo usan tanto los sistemas de archivos nativos de Linux, basados en disco, como el sistema de archivos en red NFS. La cache de buffers y de páginas interactuan íntimamente; para colocar una página de datos leída en la cache de páginas hay que pasar temporalmente por la cache de buffers. La *cache de intercambio o swap* es utilizada para mantener en memoria aquellas páginas compartidas por diferentes procesos que se han de almacenar en el área de intercambio desde la tabla de páginas de un proceso. En el momento que se ha de enviar una página compartida al área de swap, se copia dicha página en el área de

swap y, sin ser eliminada de memoria principal, se pone la página en la cache de swap. De esta manera, los restantes procesos que comparten dicha página la continúan teniendo en su espacio de direcciones. En el momento que la página es intercambiada por todos los restantes procesos será eliminada de la cache de swap y, por tanto, de la memoria principal.

4.1.3.2. Gestión de memoria virtual

El sistema de memoria virtual de Linux se encarga de mantener el espacio de direcciones visible para cada proceso. Este sistema crea páginas de memoria virtual bajo demanda y gestiona las cargas de dichas páginas desde disco o bien, en caso necesario, su intercambio a disco. Bajo Linux, el administrador de memoria virtual mantiene dos visiones distintas del espacio de direcciones de un proceso: como un conjunto de regiones individuales y como un conjunto de páginas.

La primera vista de un espacio de direcciones es la lógica. El espacio de direcciones consiste en un subconjunto de regiones que no se solapan, cada una de las cuales representa un subconjunto continuo, alineado por páginas, del espacio de direcciones. Cada región se describe internamente con una sola estructura *vm_area_struct*, que define las propiedades de la región, incluidos los permisos de lectura, escritura y ejecución del proceso en la región, e información acerca de los archivos asociados a la región.

El núcleo también mantiene una segunda vista (física) de cada espacio de direcciones. Esta vista se almacena en las tablas de páginas de cada proceso. Las entradas de las tablas de páginas determinan la ubicación exacta de cada página de memoria virtual, esté en disco o en la memoria física. La gestión se realiza por medio de un conjunto de rutinas que se invocan desde los controladores de interrupciones del núcleo, cada vez que un proceso trata de acceder a una página que no está actualmente presente en la tabla de páginas. Cada *vm_area_struct* contiene un campo que apunta a una tabla de funciones que implementan las tareas de gestión de páginas clave para cualquier región de memoria virtual dada.

Una tarea importante del subsistema de memoria virtual es el mecanismo de intercambio de páginas, el cual se encarga de reubicar páginas de la memoria física al disco (área de swap) cuando se necesita aumentar el número de páginas libres de memoria principal. El área de swap es una partición de disco que el núcleo utiliza como una extensión de la memoria principal. Linux permite definir más de un área de swap y cada una está formada por una secuencia de *slots de páginas* que pueden contener una página intercambiada. Para cada área de swap, el núcleo define una estructura denominada *swap_info_struct* que mantiene la información del estado del área para ges-

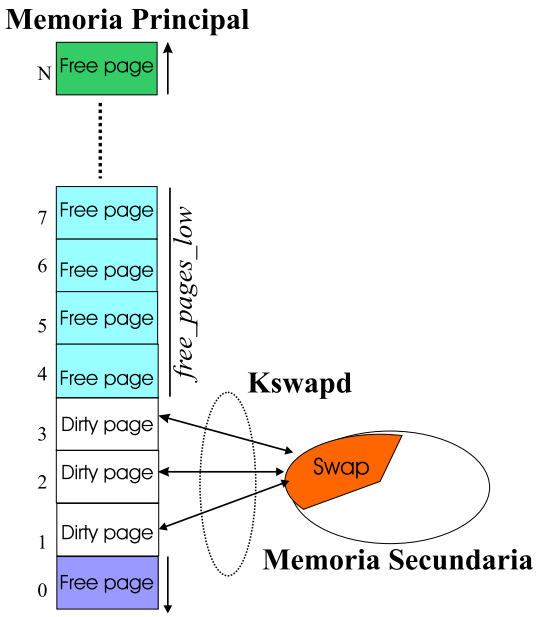

Figura 4.4: Gestión de memoria virtual.

tionar, como por ejemplo, el número de slots libres y ocupados. El núcleo dispone de un demonio de intercambio, denominado *kswapd*, que se encarga de gestionar el área de swap, así como de mantener suficientes páginas libres en memoria principal para cubrir las necesidades del sistema (ver figura 4.4). El demonio es llamado cada segundo con objeto de comprobar que haya suficientes páginas libres en la memoria principal. *Kswapd* comprueba que el número de páginas libres esté por encima de una constante denominada *free_pages_low* (=48 páginas). En caso de que el número de páginas libres esté por debajo de dicho umbral, el daemon procederá a intercambiar páginas de memoria a swap hasta que el número de páginas libres esté por encima de un segundo umbral, denominado *free_pages_high* (=144 páginas). El algoritmo de reemplazo de páginas aplicado por el daemon sigue el orden siguiente:

1. **Reducir el tamaño de la cache de páginas, de buffers y de swap**. En este caso se examina un número prefijado de páginas de la memoria principal. Aquellas páginas examinadas y que pertenecen a una de las tres caches mencionadas serán descartadas aplicando un *algoritmo del reloj o de segunda oportunidad*. Este algoritmo funciona como una lista FIFO, excepto que una vez que se ha seleccionado una página, se examina su flag de referencia asociado. Si este flag es 0, se

procederá a reemplazar la página, pero si es 1, se le dará una segunda oportunidad, de manera que se seleccionará la siguiente página FIFO. Cuando se da a una página una segunda oportunidad, su bit de referencia se pone a 0. De este modo, una página a la que se se le dio una segunda oportunidad no será reemplazada hasta que todas las demás páginas se reemplacen (o reciban una segunda oportunidad). Asimismo, si una página se usa con la suficiente frecuencia como para mantener encendido su bit de referencia, nunca será reemplazada. Si una vez aplicado dicho algoritmo, las páginas libres todavía no llegan al umbral de *free_pages_high*, el *kswapd* procederá a aplicar el punto (2).

2. **Enviar páginas de memoria compartida a swap**. El mecanismo de memoria compartida permite intercambiar información entre dos procesos mediante un conjunto de páginas de memoria principal. El demonio *kswapd*, al igual que en el caso anterior, examina un número prefijado de páginas pertenecientes al segmento de memoria compartida aplicando el algoritmo del reloj. En caso de que las páginas libres no lleguen al umbral de *free_pages_high*, el *kswapd* procederá a aplicar el punto (3).

3. **Intercambiar páginas de procesos**. En primer lugar,el *kswapd* selecciona el proceso candidato a intercambiar páginas. Este proceso será aquel que tenga un mayor requerimiento de memoria principal. Una vez seleccionado, el *kswapd* procederá a intercambiar páginas de dicho proceso aplicando el algoritmo del reloj anteriormente explicado. En caso de que la página seleccionada no hubiese sido modificada, simplemente se descarta, mientras que si había sido modificada es enviada al área de swap.

4.1.4. Subsistema de comunicaciones del s.o. Linux

La comunicación entre procesos PVM se realiza mediante *sockets*. Los sockets representan los extremos de un enlace entre dos procesos que se comunican entre si. En esta sección se describirán las diferentes capas del subsistema de comunicación del s.o. que atraviesa un mensaje, en su envío o recepción (ver figura 4.5), desde el socket de emisión hasta el socket de recepción.

El primer nivel de comunicación es la capa *sockets BSD*. Ésta es la responsable de crear el socket, así como de definir un conjunto de operaciones necesarias para que el resto de capas puedan acceder a dicho socket. Esta capa representa cada socket mediante una estructura tipo *socket*. Cada socket tiene asignado un descriptor de fichero, de manera que se convierten en

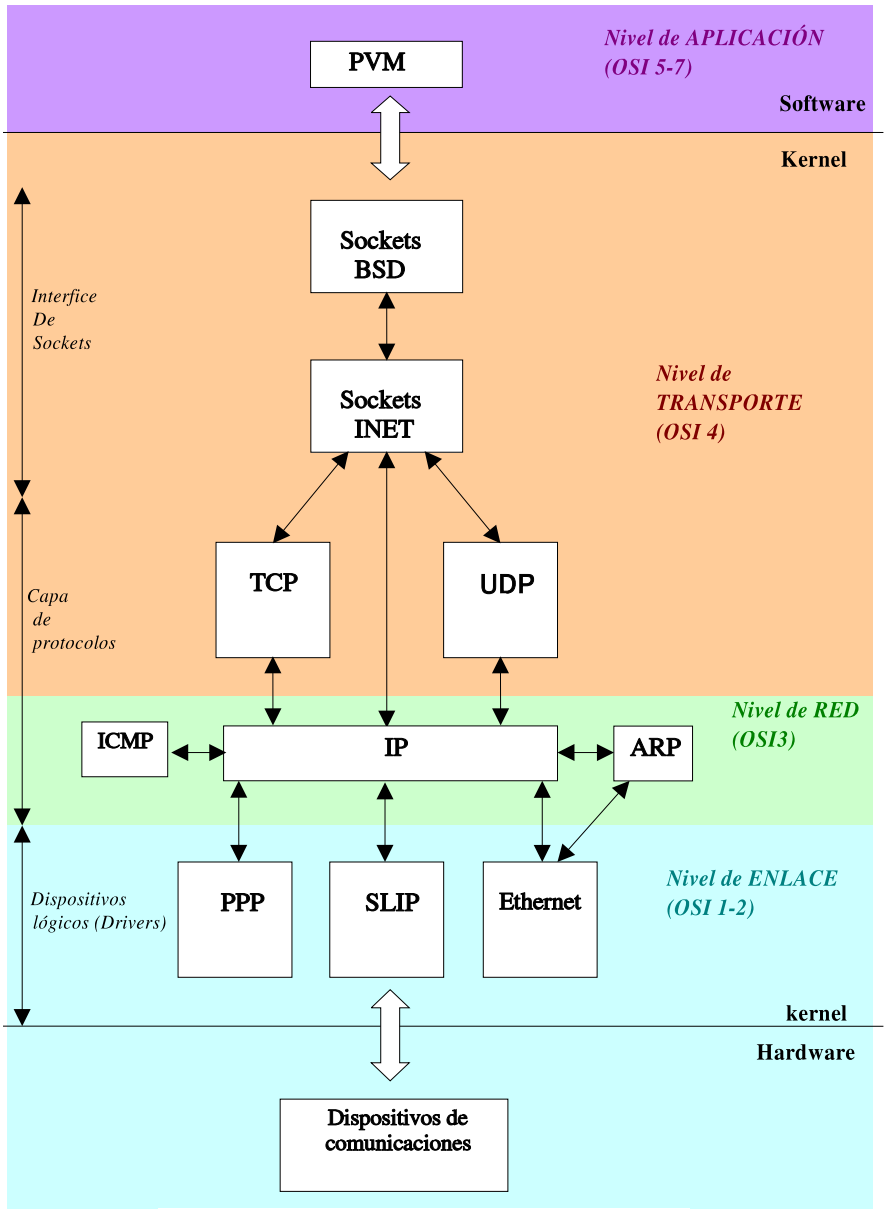

Figura 4.5: Niveles de comunicación de Linux.

un archivo más del sistema de ficheros, permitiendo a los procesos tratar por igual tanto a los sockets como a los archivos.

La siguiente capa, debajo de la capa BSD, es la capa *socket INET*. Esta capa asocia a cada estructura *socket* otra estructura de tipo *sock*. Con esta nueva estructura, la capa socket INET da significado a las operaciones que el usuario llama desde la capa socket BSD, en función de la familia de direcciones escogida. Es decir, asigna las funciones definidas por la capa BSD a las suyas propias para obtener el resultado que la capa INET desea. El conjunto de estructuras socket-sock es comunmente denominado *Socket*. Esta capa, también realiza el enrutamiento de los datos, liberando o recibiendo los datos al protocolo de red correspondiente (TCP o UDP). En el caso de trabajar con sockets de tipo *Unix Domain*, la capa de este nivel es la socket UNIX; de modo que en este caso, el flujo de datos entre los procesos implicados sólo pasará por las dos capas correspondientes a los sockets.

En la capa socket INET, los mensajes enviados por el nivel de usuario (PVM en nuestro caso) son empaquetados en paquetes de tamaño fijo (*MTU: "Maximum Transmission Unit"*). Asimismo, en el caso de una recepción, esta capa se encarga de convertir los paquetes en mensajes. Cada paquete, en el kernel, es representado mediante una nueva estructura de datos, denominada *sk_buff*. Los sk_buffs son el medio de transporte que usan los datos (paquetes) para atravesar el kernel. Un sk_buff es una estructura de datos compuesta por unos cuantos campos de control y un bloque de datos asociados. La gestión de los sk_buffs se lleva a cabo mediante un conjunto específico de funciones definidas por el núcleo. La figura 4.6 muestra el esquema básico de esta estructura y el conjunto de campos que proporcionan el control sobre los datos contenidos en el buffer. De este modo, en la emisión de un paquete, la capa socket INET creará un nuevo sk_buff, el cual será almacenado en una cola denominada *write_queue*; mientras que en la recepción, los paquetes recibidos serán almacenados en una cola denominada *receive_queue*. Ambas colas tienen un tamaño máximo de 64KBytes.

La siguiente capa corresponde al tipo de protocolo que se establece en la creación del socket, que puede ser del tipo *TCP* (*"Transmision Control Protocol"*) o bien *UDP* (*"User Datagram Protocol"*). La capa TCP implementa un tipo de protocolo con control de transmisión para enlaces seguros, mientras que UDP implementa un protocolo no seguro de transmisión y recepción de paquetes.

A continuación se encuentra la capa de red. Está formada por, entre otros, el protocolo *IP* ("*Internet Protocol*"), que proporciona un servicio de transmisión/recepción de paquetes. Éste se ayuda del protocolo *ARP* en la resolución de las direcciones físicas de los nodos destino. El *ICMP* ("*Internet Control Messages Protocol*") es el mecanismo usado para la notificación de mensajes de error y está encapsulado dentro de las tramas IP. En la emisión o recepción de paquetes, cada uno de los protocolos descritos añadirá o sacará

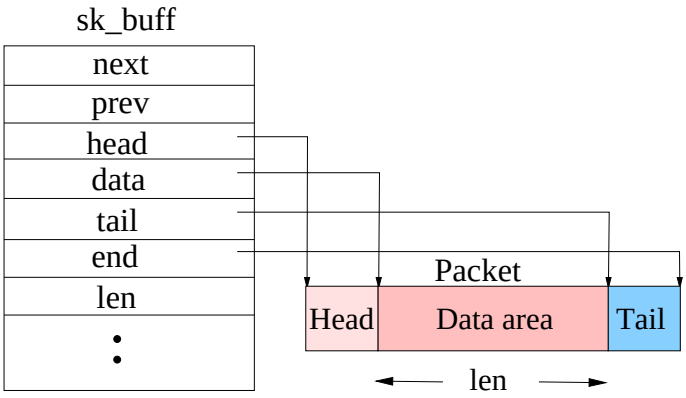

Figura 4.6: Estructura sk_buff.

información de control de los campos *Head* y *Tail* de la estructura sk_buff, mostrada en la figura 4.6.

Por debajo de la capa IP se encuentra la interficie con los dispositivos de red. Esta capa representa cada dispositivo mediante una estructura *device*, la cual proporciona la información y las operaciones necesarias para poder comunicarse con los dispositivos físicos. En esta capa, los sk_buff de transmisión son almacenados en una de tres posibles colas: *interactive* (alta prioridad), *normal* (mensajes PVM) y *background* (baja prioridad); teniendo todas ellas una capacidad de 100 elementos. La elección de una u otra cola depende de la prioridad del paquete. Por otro lado, los paquetes recibidos son almacenados en una cola, denominada *backlog*, la cual tiene una capacidad máxima de 300 sk_buffs.

Los dispositivos físicos forman el ultimo escalón en la estructura de capas y su misión es transformar los datos para que puedan viajar por el medio físico.

4.1.5. El sistema PVM

PVM ("*Parallel Virtual Machine*") [GBD+94] permite trabajar con una red de máquinas heterogéneas como si de una máquina paralela se tratara, permitiendo, de una manera totalmente transparente para el programador, el intercambio de mensajes, las conversiones de formatos y la planificación de las tareas a lo largo de la red.

El sistema PVM está compuesto por dos partes bien diferenciadas. La primera de ellas es el *daemon*, llamado *pvmd*, que reside en todos los computadores que forman la máquina virtual. Este daemon, básicamente, gestiona la

comunicación entre las tareas PVM, permitiendo la ejecución concurrente de las mismas, en los diferentes nodos que forman la máquina virtual paralela. La segunda parte del sistema es la *librería PVM*, denominada *pvmlib*, formada por un conjunto de funciones C/Fortran que permiten el intercambio de mensajes, la activación de tareas remotas, la sincronización entre tareas, así como la modificación de la configuración de la máquina virtual; de manera que una tarea PVM estará compuesta por procesos UNIX enlazados con la librería *pvmlib*.

PVM utiliza un identificador de tareas (*TID*) para direccionar los daemons, tareas y grupos de tareas dentro de la máquina virtual. Los intercambios de mensajes dentro de un sistema PVM se pueden clasificar en mensajes de sistema, utilizados por PVM para controlar la máquina virtual, y los mensajes propios de las aplicaciones.

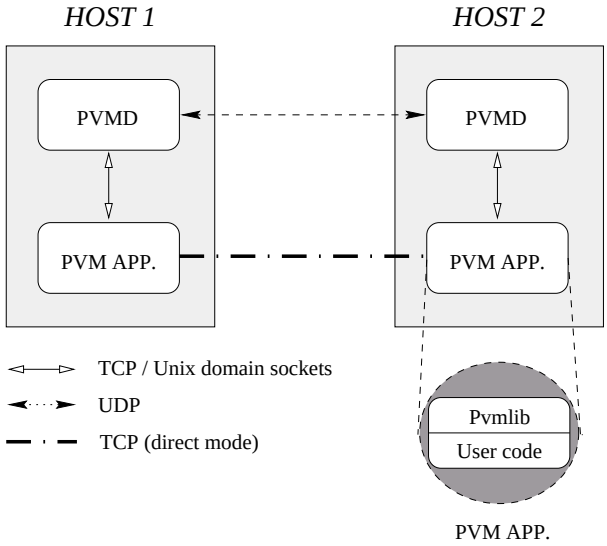

Figura 4.7: Sistema PVM.

Las comunicaciones en PVM se realizan mediante sockets, de manera que PVM ofrece dos mecanismos de enrutamiento para los mensajes de las aplicaciones: directo e indirecto. La elección de uno u otro es controlado por el mismo código de la aplicación. Usando el mecanismo de enrutamiento indirecto, como ilustra la figura 4.7, un mensaje desde una tarea del host1 a una tarea del host2, es enviado a través de los daemons (*pvmd*) del host1 y host2. La comunicación entre los daemons se realiza mediante el protocolo UDP, mientras que la comunicación entre tarea-pvmd se realiza mediante el protocolo TCP o bien, a partir de la versión 3.3 de PVM, a través de sockets

de dominio Unix. En el mecanismo directo, las comunicaciones entre tareas se realizan sin la colaboración de los daemons, utilizando el protocolo TCP. En resumen, los protocolos de transporte usados en PVM son:

- **pvmd-pvmd**. Protocolo UDP.
- **tarea-pvmd**. Protocolo TCP o por defecto, a partir de la versión 3.3, sockets de dominio UNIX .
- **tarea-tarea**. De manera directa, una tarea envía sus mensajes a otra en distinta máquina, usando el protocolo TCP. Por defecto se realiza de forma indirecta, pudiéndose pasar a modo directo mediante la rutina *PvmRouteDirect()* que implementa la librería *pvmlib*.

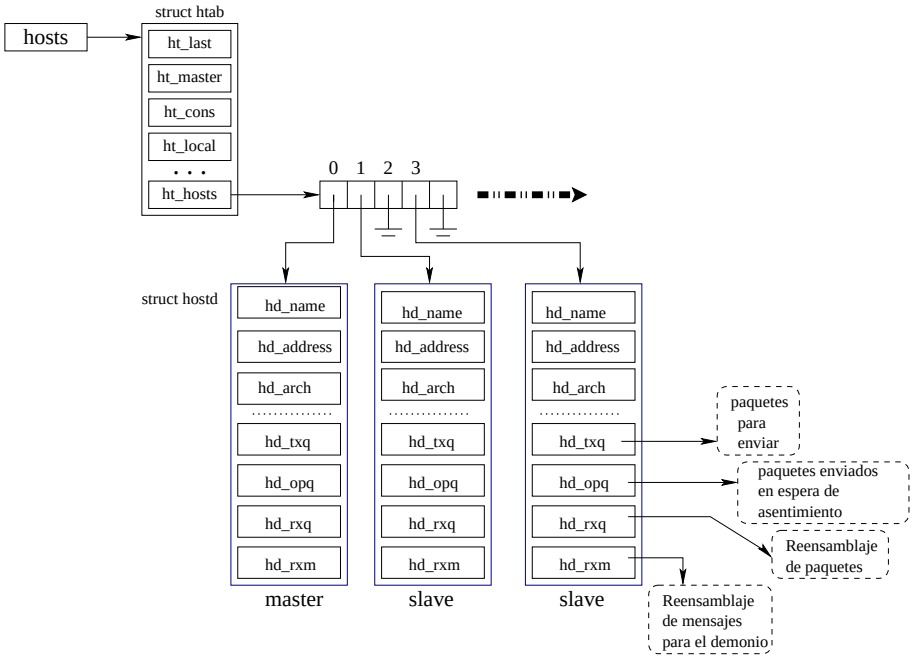

Figura 4.8: Tabla de hosts PVM activos.

Cada daemon local mantiene una tabla denominada *"host table"*, la cual describe la configuración de la Máquina Virtual Paralela (MVP). La figura 4.8 muestra un esquema de dicha tabla. En esta figura se observa como la tabla de hosts está formada por una lista de estructuras *hostd,* una por cada nodo que compone la *MVP,* que proporcionan a cada demonio la información necesaria para poder comunicarse con el resto de hosts que componen

la MVP: dirección del socket ($hd\_address$), tipo de arquitectura (hd_arch), nombre (hd_name), etc... Es importante destacar que esta lista de estructuras realiza la misma funcionalidad que la lista *cooperating* descrita en el modelo de cluster no dedicado definido en este trabajo de tesis (ver sección 2.1). Mediante esta lista, cada nodo almacena las direcciones del resto de nodos del cluster donde se encuentran el resto de tareas remotas. Asimismo, cada una de estas estructuras *hostd* contiene un conjunto de colas donde son insertados todos paquetes recibidos desde los otros *pvmds* remotos, o bien aquellos paquetes en espera a ser transmitidos a otros *pvmds*.

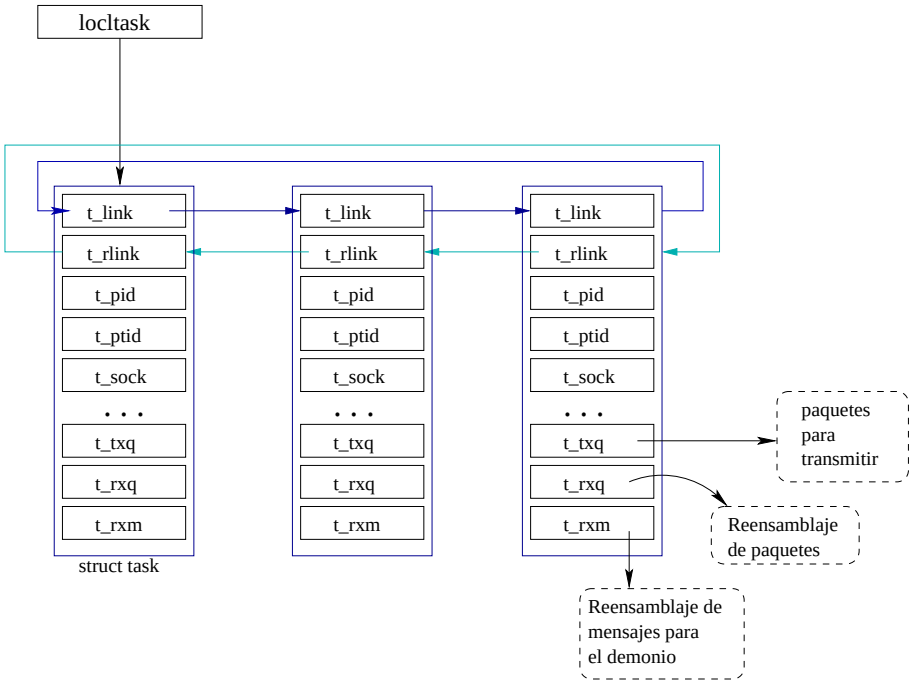

Figura 4.9: Tabla de tareas PVM locales.

Asimismo, cada daemon posee una lista de estructuras de tipo *task* para representar el conjunto de tareas que se encuentran ejecutando en dicho nodo. En cada una de estas estructuras, el daemon mantiene toda la información que necesita para poder comunicarse con la tarea asociada, como por ejemplo: la dirección del socket, el pid de la tarea, el estado de la tarea, las colas de envío y recepción de mensajes, etc... Esta lista será especialmente útil para implementar el algoritmo de balanceo de carga, descrito en la sección 3.5, dado que permite realizar la conversión del identificador de proceso con el que trabaja el s.o. (pid), en el correspondiente identificador de trabajo al que pertenece dicho proceso ($ptid$).

La gestión de PVM, por parte del usuario correspondiente, se realiza desde la *consola pvm*. A través de ésta, se van añadiendo las diferentes máquinas que compondrán la MVP. Cada vez que se añade una nueva máquina, se arranca el correspondiente *pvmd* en dicha máquina.

4.2. Implementación del CoScheduling Cooperativo (CSC)

CSC ha sido implementado en parte en el espacio de usuario, en concreto en el daemon del sistema PVM, así como en el propio kernel del s.o. Linux. La implementación en el espacio del kernel garantiza que CSC se adapte rápidamente a los constantes cambios requeridos tanto por el usuario local, asegurando el tiempo de respuesta de sus aplicaciones dentro de unos márgenes aceptables por el mismo, como a los requerimientos de las aplicaciones distribuidas. Asimismo, diferentes algoritmos de CSC no pueden ser implementados en el espacio de usuario debido a la ausencia de llamadas a sistema en Linux que permitan modificar aquellas estructura internas del núcleo necesarias para la implementación de dichos algoritmos. Finalmente cabe destacar que, la implementación de CSC ha supuesto la modificación o inclusión de alrededor de 250 líneas del código de kernel original, la mayor parte de ellas asociadas con la implementación de las llamadas a sistema, descritas en el apéndice C, así como la inclusión de alrededor de 100 nuevas líneas de código asociadas con el daemon de PVM.

En esta sección se describen las diferentes modificaciones que se han realizado tanto, en el kernel de Linux como en el entorno PVM, para implementar cada uno de los módulos que forman el CoScheduling Cooperativo.

4.2.1. Módulo de gestión de memoria

Con objeto de implementar el algoritmo mostrado en la figura 3.2 se ha modificado el algoritmo original de Linux para seleccionar la tarea candidata para el intercambio de páginas (función *swap_out* del kernel de Linux), descrito en la sección 4.1.3.2. El algoritmo 3 muestra el código introducido (en cursiva) con respecto al código original.

En las líneas 2, 3 y 4, el algoritmo procede a recalcular los requerimientos totales de memoria de las tareas distribuidas. El valor calculado se almacena en una nueva variable definida en el kernel para este propósito, denominada *mem_par*. A continuación, el algoritmo recorre todas las tareas con el objetivo de seleccionar la mejor candidata para intercambiar páginas (*swaptask*), aplicando exactamente el algoritmo descrito en la sección 3.2.

Algoritmo 3 Función swap_out.

```
1 swaptask:=NULL; mem_par:=0; max_vmem:=0;
2 for_each_task(p){
3    if(p->uid=PARAL) mem_par:=mem_par+p->mm->total_vm;
4 }
5 for_each_task(p){
6     if (p->state_par=STOP or p=stoptask)
7         swaptask:=p;
8     else if (mem_par>MEM_TOTAL*L)
9         if (p->mm->total_vm>max_vmem and p->uid=PARAL){
10            swaptask:=p;
11            max_vmem:=p->mm->total_vm;
12        }
13    else
14        if(p->mm->total_vm>max_vmem){
15            swaptask:=p;
16            max_vmem:=p->mm->total_vm;
17        }
18  }
19 swap_out_processs(swaptask);
20  if (swaptask->uid=PARAL and LOCAL_USER )
21    stoptask:=swaptask;
```

La aplicación de dicho algoritmo ha comportado crear una nueva llamada, denominada *heterogeneity()*, que permite pasar al kernel el porcentaje de recursos asignados a las tareas paralelas (L). El código asociado a esta llamada se muestra en el apéndice C. Con objeto de que en todos los nodos del cluster, las tareas paralelas dispongan de la misma porción de memoria disponible, independientemente del tamaño de la memoria principal, esta porción (MEM_TOTAL*L) ha sido calculada respecto al menor tamaño de memoria principal presente a lo largo del cluster (MEM_TOTAL). El valor de MEM_TOTAL es pasado al kernel de cada nodo por medio de la misma llamada a sistema *heterogeneity()*. Asimismo, un nuevo campo, denominado *state_par*, ha sido añadido a la estructura de procesos *task_struct*. Este campo mantiene el estado (STOP, LOCAL o NULL) de la aplicación paralela a la que pertenece *task* y se corresponde con el campo *state* definido en el modelo de coscheduling descrito en la sección 2.1. Una vez seleccionada la tarea *swaptask*, el algoritmo llama a la función del kernel *swap_out_process(swaptask)* encargada de realizar el intercambio de páginas sobre la tarea pasada como parámetro. Finalmente, en caso de que en dicho nodo haya un usuario local (*LOCAL_USER=1*), el algoritmo procede a asignar el puntero *stoptask* a

swaptask, con objeto de que el bloque de asignación de quantum detenga la misma tarea que ha sido seleccionada para el swapping.

4.2.2. Módulo de gestión de la cola de preparados

En esta sección se describe como se ha implementado el módulo de gestión de la cola de preparados, descrito en la sección 3.3, en el kernel de Linux. Con objeto de facilitar la comprensión del mismo, se ha seguido el mismo orden utilizado en la sección 3.3. En primer lugar se describe la gestión de la cola de preparados realizada en aquellos nodos con bajos requerimientos de memoria. Posteriormente, en la sección 4.2.2.1 se describe el algoritmo aplicado en los nodos con altos requerimientos de memoria.

En aquellos nodos donde las tareas encajan en memoria, la prioridad de planificación de cada tarea es incrementada de acuerdo con los mensajes recibidos y enviados por cada proceso. Por este motivo, en primer lugar se ha implementado el mecanismo para capturar estos dos valores y almacenarlos en dos nuevos campos, creados para este propósito, en la estructura de tareas (*task_struct*). Estos dos nuevos campos son denominados *packr* y *packs*, respectivamente.

Estos campos han sido calculados por medio de una nueva función implementada en el kernel, denominada *n_packets(task,queue).* Dependiendo del valor del argumento *queue*, esta función retorna el número de paquetes en la cola de sockets de recepción (*receive_queue*) o de transmisión (*write_queue*). El pseudo-código de dicha función se muestra en el algoritmo 4.

Algoritmo 4 Función n_packets(task,queue).

```
1  if (task→files)
2    for (fd = 0; fd < task→files→max_fds; fd++) {
3      file := task→files→fd[fd];
4      if (file) {
5        inode := file→f_dentry→d_inode;
6        if (inode and inode→i_sock and socket := socki_lookup(inode))
7          if (queue = receive_queue)
8             task→packr+ :=⌈socket→sk→rmem_alloc.counter/4096⌉;
9          else task→packs + := ⌈socket→sk→wmem_alloc.counter/4096⌉;
10     }
11 }
12 if (queue = receive_queue) return packr; else return packs;
```

En Linux, los sockets son tratados como ficheros. Por tanto, en primer

lugar se debe identificar aquellos ficheros que se correspondan con los sockets. Con este objetivo, se accede a la estructura que representa a cada fichero en Linux, la estructura *inode*. Como se puede ver en la figura 4.10, el acceso a *inode* se realiza descendiendo a partir de una cascada de estructuras, de acuerdo con el siguiente orden: task_struct ->files_struct ->file ->dentry ->inode. De este modo, si el inode corresponde a un socket (condición *inode->i_sock* del algoritmo 4) se podrá acceder a la estructura socket asociada (*socket=socki_lookup(inode)*), por medio de la función interna del kernel *socki_lookup*. Finalmente, desde la estructura *socket* se podrá acceder a la estructura *sock* que apunta a las dos colas buscadas (*receive_queue* y *write_queue*). Los campos *rmem_alloc* y *wmem_alloc* de la estructura *sock* contienen el número de bytes en las colas *receive_queue* y *write_queue*, respectivamente.

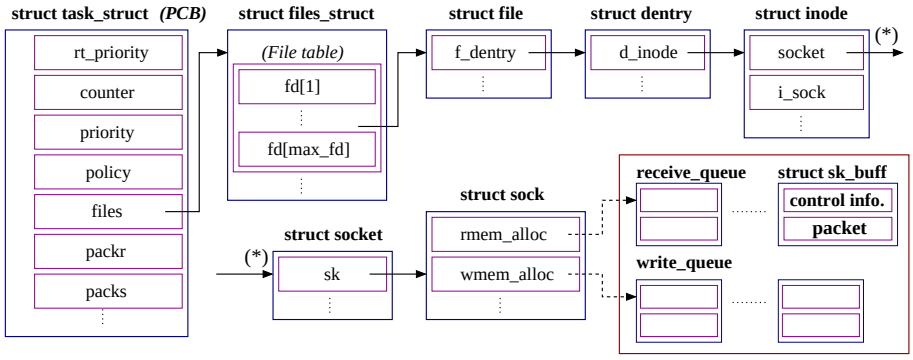

Figura 4.10: Estructuras Linux involucradas en la lectura de paquetes recibidos y enviados.

Con objeto de actualizar los campos *packr* y *packs* asociados a cada tarea distribuida, la función *n_packets* es llamada en los siguientes puntos del kernel:

- Los paquetes recibidos por *task* (*n_packets(task,receive_queue)*) son leídos cada vez que *task* es insertada en la cola de preparados, por medio de la función interna del kernel *add_to_run_queue*().
- Los paquetes transmitidos por *task (n_packets(task,write_queue))* son leídos cada vez que *task* finaliza su tiempo de CPU, es decir en el punto 2 del algoritmo de planificación de Linux explicado en la sección *4.1.2.*

Finalmente, con objeto de asignar la prioridad de planificación de las tareas en función de los mensajes recibidos y transmitidos, la función *goodness* del

planificador de Linux ha sido modificada. El algoritmo 5 muestra el pseudocódigo de la función *goodness* (las líneas añadidas para la implementación del coscheduling dinámico se muestran en cursiva).

Algoritmo 5 Función *goodness*(*task*).

```
1  if (task→policy ! = SCHED_OTHER) //   "real" priority
2    weight := task→rt_priority + 1000;
3  else { //    "normal" priority
4    weight := task→counter + task→priority;
5    weight +:= task→packr+ task→packs;
16 }
12 return weight;
```

En el algoritmo original de *goodness*, si una tarea es de tiempo real, su prioridad de planificación (*weight*) es asignada como mínimo a 1000; en caso contrario, su prioridad es asignada de acuerdo con los valores de los campos *counter* y *priority*. En nuestra implementación y de acuerdo con la filosofía del coscheduling dinámico, la prioridad de las tareas de tiempo compartido ha sido incrementada en función de los paquetes recibidos (*packr*) y transmitidos (*packs*).

Un aspecto que no se ha implementado, respecto al algoritmo de coscheduling descrito en la sección 3.3, es la limitación en el número de adelantos sufridos por una tarea en la cola de preparados. Este mecanismo no ha sido implementado dado que el propio algoritmo de reasignación del quantum de Linux, (ver algoritmo 2), ya incorpora un mecanismo para favorecer la planificación rápida de las tareas interactivas, propias de un usuario local.

Asimismo, con objeto de distinguir entre tareas competidoras homogéneas pertenecientes a distintos trabajos distribuidos, se ha modificado el *algoritmo de selección del proceso* a planificar, descrito en la sección 4.1.2, de acuerdo con la ecuación 3.7, descrita en la sección 3.3.1. Con este fin, al algoritmo de selección original se han incorporado las líneas 9 y 10 mostradas en el algoritmo 6. De acuerdo con nuestra propuesta, la modificación realizada prioriza aquella tarea distribuida que hace menos tiempo que fue lanzada al sistema; es decir aquella que tiene un *start_time* mayor.

4.2.2.1. Coscheduling con restricciones de memoria

En aquellos nodos donde la memoria es desbordada se aplica el algoritmo descrito en la sección 3.3.2, que prioriza aquellas tareas con menor tasa de

Algoritmo 6 Selección del proceso a planificar de acuerdo con el algoritmo de coscheduling entre tareas distribuidas homogéneas.

```
1  c := -1000;
2  p := init_task->next_run;
3  while(p != init_task){
4    weight := goodness(p);
5    if(weight>c){
6      c := weight;
7      next: = p;
8    }
9    else if(weight=c and p->start_time>next->start_time and p->uid=PARAL)
10         next := p;
11   p =p->next_run;
11 }
```

fallos de página. La implementación de este algoritmo ha sido realizada de acuerdo con los siguientes pasos:

1. **Consulta de la cantidad de memoria principal libre**. El algoritmo de coscheduling con restricciones de memoria se activa cuando los requerimientos de memoria de los procesos residentes en dicho nodo superan los límites de la memoria principal. Esta situación es detectada por medio de la variable interna del núcleo, *nr_free_pages*, que contabiliza en todo momento el número de páginas libres de memoria. Tomando los mismos límites que utiliza el daemon de swapping (*kswapd*) para intercambiar páginas, este algoritmo se activará siempre y cuando se cumpla la siguiente condición:

$$nr\_free\_pages < free\_pages\_low, \tag{4.1}$$

   siendo *free_pages_low* una constante determinada por el mismo kernel ($= 48\, paginas$).

2. **Cálculo de la ratio de fallos de página (flt)**. La ratio de fallos de página de cada proceso ha sido calculada a partir de los fallos de página con acceso a disco (*maj_flt*), proporcionados por el mismo sistema operativo a partir de la estructura de tareas (*task_struct*) de cada proceso. De este modo, cada vez que el daemon de swapping (*kswpd*) es llamado ($=1sg$), comprobará la cantidad de memoria libre por medio de la evaluación de la condición 4.1 y, en caso de que ésta sea cierta, el mismo daemon procederá a recalcular, de acuerdo con el algoritmo 7, los fallos de página mayores realizados desde la última vez que se ejecutó el daemon por cada uno de los procesos distribuidos. Esta ratio será

almacenada en un nuevo campo añadido a la estructura de tareas para este propósito (*flt*). Este algoritmo ha sido implementado en la función *swap_ out()* del kernel, descrita anteriormente en la implementación del bloque de gestión de memoria.

Algoritmo 7 Cálculo de la ratio de fallos de página.

```
1 temp_flt := 0;
2 for_each_parallel_task(p){
3   p-> flt := p-> maj_flt - p-> maj_flt_old;
4   p-> maj_flt_old := p-> maj_flt;
5   if(p-> flt > temp_flt)   temp_flt := p-> flt;
6 }
```

3. **Planificación de los procesos de acuerdo con su ratio de fallos de página**. Con este objetivo se ha modificado la función *goodness*() del planificador, de acuerdo con el algoritmo 8. En cursiva se muestran las nuevas líneas añadidas. Este algoritmo muestra como la ratio de fallos de página de cada proceso es ponderada en función de la ratio mayor (*temp_flt*), calculada en el paso previo (ver algoritmo 7); de modo que el proceso con una mayor ratio de fallo tendrá su prioridad disminuida en 10 créditos; factor que puede ser considerado alto si se tiene en cuenta que la prioridad por defecto de todos los procesos de tiempo compartido (*DEF_PRIORITY*) es igual a 21 créditos.

Algoritmo 8 Función *goodness*(*task*) modificada de acuerdo con el algoritmo de coscheduling con requerimientos de memoria.

```
1 if (task→policy != SCHED_OTHER) //   "real" priority
2   weight := task→rt_priority + 1000;
3 else { //    "normal" priority
4   weight := task→counter + task→priority;
5   if (nr_free_pages≥free_pages_high)
6       weight +:= task→packr+ task→packs;
7   else
8       weight -:= (task→flt/temp_ maj_flt)*10;
9 }
10 return weight;
```

4.2.3. Módulo de gestión del quantum

La implementación del módulo de gestión de quantum, descrito en la sección 3.4, ha comportado las siguientes etapas:

1. **Detección del usuario local**. La detección de la presencia del usuario local se realiza mediante la consulta de la actividad del teclado y del ratón. El planificador de Linux, cada vez que es llamado, comprueba la existencia de interrupciones pendientes (*bottom_ halves*) del teclado y del ratón (ver sección 4.1.2). En el caso de que el flag asociado con los *bottom_ halves* del teclado y ratón estén activados, una nueva variable booleana del núcleo, denominada *LOCAL_ USER*, será puesta a 1. Esta variable tiene asociado un campo de tiempo, donde se almacena el instante en el cual se ha producido la última actualización. La variable *LOCAL_ USER,* junto con su referencia temporal, es comprobada al final de cada época del planificador, de modo que si no se ha detectado ninguna actividad de teclado o ratón en más de 1 minuto la variable *LOCAL_ USER* será puesta a 0. El umbral de un minuto ha sido escogido teniendo en cuenta los estudios de Mutka y Livny [ML91], donde se muestra experimentalmente como este valor asegura una alta probabilidad de que el usuario local haya finalizado su trabajo y, al mismo tiempo, conlleva que el sistema no desperdicie los recursos de cómputo libres disponibles.

2. **Heterogeneidad del procesador**. El tratamiento de la heterogeneidad del procesador ha comportado, en primer lugar, obtener la potencia de cómputo relativa (*Power Weight*) de cada uno de los nodos del cluster. El *Power Weight* ha sido calculado de acuerdo con la ecuación 3.9, descrita en la sección 3.4.2. La implementación de dicha formula ha supuesto obtener, en cada uno de los nodos del cluster, dos diferentes valores: una métrica asociada a la potencia del procesador residente en dicho nodo, así como el valor de esta misma métrica asociada con el nodo más rápido del cluster. Como medida de la potencia de procesador se han tomado los *BOGOMIPS,* métrica que retorna el mismo kernel en su proceso de inicio (función *start_ kernel()*). Este valor es calculado en la fase de inicialización del sistema por medio de la función interna *calibrate_ delay()*. Los *BOGOMIPS* asociados con el nodo más rápido han sido pasados al kernel por medio de la llamada a sistema *heterogeneity*(). Esta llamada es explicada en el apéndice C.

3. **Reanudación de la tarea stoptask.** La tarea detenida por el bloque de gestión de memoria, denominada *stoptask*, es reanudada siempre

y cuando se cumpla que la suma de los requerimientos de memoria totales sea inferior al *95 %* del tamaño de la memoria principal. Esta condición se evalúa en dos puntos diferentes del kernel: cada vez que una tarea finaliza su ejecución y por consiguiente se libera memoria (función *do_ exit()* del kernel), o bien cada vez que finaliza una época de planificación y el planificador debe reasignar un nuevo quantum (función *schedule()* del kernel).

4. **Modificación del quantum**. Una vez calculados los valores anteriormente descritos, el algoritmo de asignación de quantum en entornos con bajos requerimientos de memoria, descrito en la sección 3.4, puede ser aplicado. Con este objetivo, al final de cada época, en concreto en la etapa de *reasignación del quantum* del planificador de Linux (ver algoritmo 2), el quantum (*counter*) de cada tarea distribuida es reasignado de acuerdo con el algoritmo mostrado en la figura 3.4 del capítulo anterior.

4.2.3.1. Asignación de la longitud del quantum en entornos con altos requerimientos de memoria

La implementación del algoritmo mostrado en la figura 3.6 ha comportado la modificación de la función *swap_ out()* y del planificador de Linux (función *schedule()*). Estas modificaciones se han realizado en tres pasos:

1. **Selección del proceso a detener.** Con objeto de disminuir los requerimientos de memoria de las tareas paralelas, en aquellos nodos donde existe un usuario local, es decir cuando la variable interna *LOCAL_ USER* está activada, se ha detenido (*counter:=0*) aquella tarea paralela con mayores requerimientos de memoria (*stoptask*). La búsqueda de la tarea candidata a ser parada se ha implementado en la función *swap_ out()* del kernel, descrita en la implementación del bloque de gestión de memoria.

2. **Asignación de la constante STEP quantum**. En aquellos nodos donde la memoria es desbordada y no existe acividad de usuario local, la longitud del quantum, asignada a las tareas distribuidas, es incrementada proporcionalmente al valor de STEP. El valor de esta variable es fijado por el usuario mediante el uso de la llamada a sistema *heterogeneity()* descrita en el apéndice C.

3. **Reasignación de quantum.** Al final de cada época del planificador, en la etapa de reasignación del quantum del planificador de Linux (fun-

ción *schedule()*), el quantum (*counter*) de cada tarea distribuida ha sido reasignado de acuerdo con el algoritmo de la figura 3.6.

4.2.4. Módulo de gestión de recursos remotos

El algoritmo de balanceo de recursos ha sido implementado en parte en el espacio de kernel y el resto en el propio *daemon de PVM*, aprovechando la información que ofrece el mismo daemon de PVM a partir de sus estructuras internas. Como se describió en la sección 3.5, este algoritmo de balanceo de recursos se basa en el intercambio, entre nodos cooperantes, de aquellos eventos que han comportado un cambio en la cantidad de recursos de cómputo asignados a las tareas paralelas. La figura 4.11 muestra los pasos que comporta el envío y la recepción de un evento:

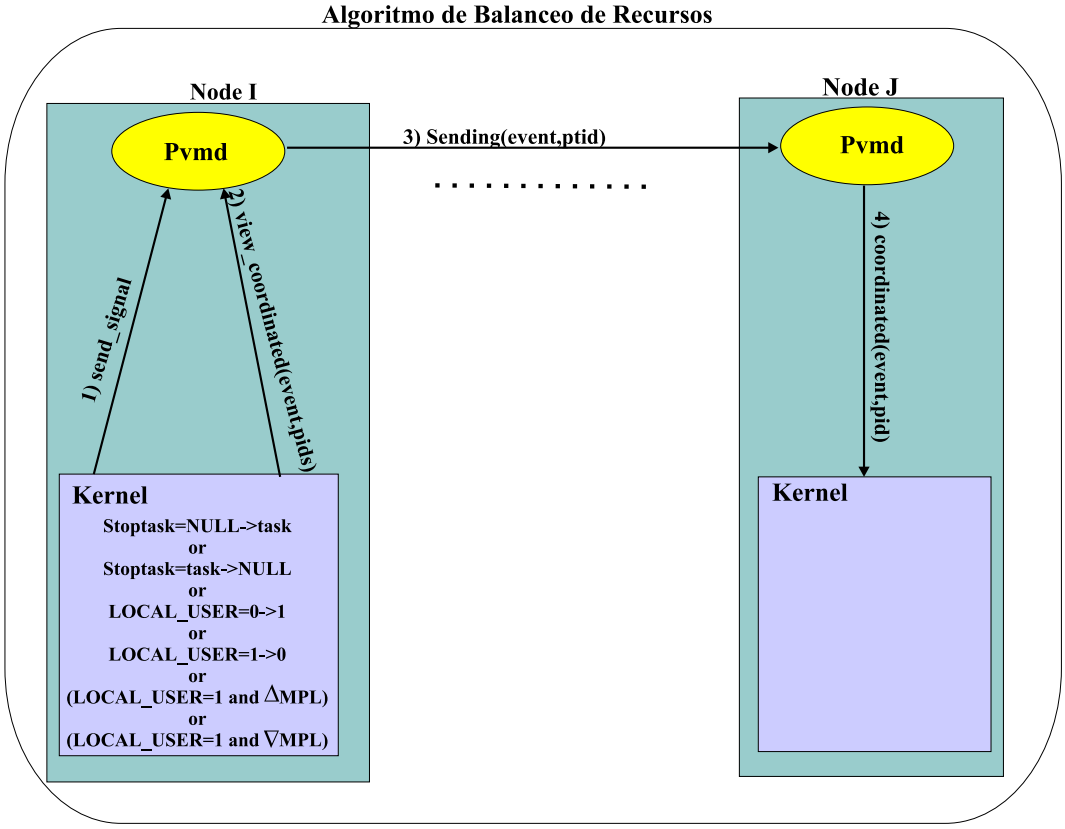

Figura 4.11: Implementación del algoritmo de balanceo de recursos.

1. En primer lugar, el algoritmo debe detectar cuando se ha producido una reasignación de recursos en un determinado nodo. Con este fin,

el kernel evalúa al final de cada época, las seis condiciones mostradas en la figura 4.11. Exceptuando las dos últimas condiciones, las cuatro primeras dependen del valor de las variables *stoptask* y *LOCAL_ USER,* explicadas en las secciones precedentes. La evaluación de las dos últimas condiciones comporta el cálculo de un nueva variable, denominada *MPL*, cuyo objetivo es contar el número de procesos paralelos que se ejecutan concurrentemente en cada instante de tiempo. Este contador, denominado *MPL*, se incrementa cada vez que se crea un nuevo proceso perteneciente al usuario distribuido (*task->uid*=PARAL), ya sea desde la función *do_fork()* o bien desde la función *do_ exec()* del kernel. Cabe recordar que con objeto de facilitar la detección de las tareas distribuidas, todas las tareas distribuidas se ejecutan con un mismo identificador de usuario *(PARAL).* Asimismo, este mismo contador se decrementa cada vez que un proceso finaliza su ejecución. Este decremento se realiza desde la función *do_ exit()* del kernel. Una vez evaluadas estas seis condiciones y, en caso de que alguna de ellas haya sido activada, el kernel, con objeto de informar al resto de nodos cooperantes de la variación de recursos asignados a una o varias tareas distribuidas, realiza la notificación de la ocurrencia del evento asociado con la condición activada mediante el envío, al *daemon de PVM,* de una señal (*SIGUSR1*). Este envío se realiza mediante la función propia del kernel *send_ sig()*. Dado que dicha función necesita el *pid* del proceso destino de la señal, el *pid* del daemon de PVM ha sido capturado desde el propio espacio de kernel. Esta captura se realiza desde la función *do_ exec()* del kernel.

2. Una vez el daemon de PVM ha recibido la señal enviada por el kernel, éste ejecuta la llamada a sistema *view_ coordinated()* (ver apéndice C), la cual retorna el tipo de evento ocurrido, así como el *pid* asociado a cada una de las tareas distribuidas PVM afectadas por la ocurrencia de dicho evento.

3. En el tercer paso, el daemon de PVM convierte cada uno de los *pids* recibidos desde el kernel en los correspondientes identificadores de los trabajos PVM *(ptids*), a los cuales pertenecen las tareas notificadas. Esta conversión se realiza mediante el acceso a la estructura de PVM *task*, explicada en la sección 4.1.5, que contiene la información asociada con cada tarea PVM. Una vez realizada dicha conversión, el daemon de PVM procederá a enviar un mensaje en *multicast* al resto de nodos, donde residen las tareas cooperantes asociadas con la tarea notificada. Las direcciones de los nodos cooperantes son leídas a partir de las

estructuras del daemon *hostd*, asociadas a cada uno de los nodos que constituyen la máquina virtual. En este mensaje, el daemon de PVM, enviará dos diferentes parámetros:

- *evento*: El código asociado al evento notificado.
- *ptid*: Identificador de la aplicación a la cual pertenece la tarea notificada.

4. Una vez el daemon de PVM receptor, ha recibido la notificación del evento correspondiente, realizará la conversión inversa a la realizada en el punto dos. De este modo, el daemon de PVM, a partir del *ptid* recibido, obtendrá el *pid* asociado con la tarea cooperante notificada. A continuación, el daemon procede a modificar los recursos de cómputo de la tarea notificada de acuerdo con el evento recibido. Esta modificación se realiza mediante la ejecución de la llamada a sistema, denominada *coordinated()*, implementada para este fin. Cabe decir que el algoritmo de recepción de eventos, mostrado en la figura 3.8 ha sido implementado en la misma llamada a sistema *coordinated()*. Esta llamada es descrita en el apéndice C.

En conclusión, la implementación de CSC descrita en este capítulo, desde el nivel del s.o. hasta el nivel de usuario de las aplicaciones distribuidas, ha permitido que CSC pueda ser instalado en un entorno cluster real con el objetivo de evaluar la viabilidad del mismo, con respecto a las políticas de coscheduling tradicionales más significativas. Esta evaluación será descrita en el siguiente capítulo.

Capítulo 5

Evaluación del Rendimiento de CSC en un Entorno NOW

En este capítulo se evalúa el comportamiento de la implementación realizada de CSC en diferentes ámbitos, caracterizados por diferentes requerimientos tanto de las aplicaciones locales como de las distribuidas. De este modo, el rendimiento de CSC es analizado tanto en lo que respecta a la percepción del usuario local como al punto de vista del usuario de las aplicaciones distribuidas.

La experimentación ha sido realizada en dos diferentes tipos de NOW: un *entorno NOW controlado* y un *entorno NOW de producción*. El escenario controlado se caracteriza porque los requerimientos de la carga local han sido simulados mediante el uso de un benchmark sintético, el cual permite, al usuario del mismo, fijar tanto los requerimientos de comunicación, memoria, E/S a disco y CPU. Estas pruebas han permitido sintonizar los diferentes parámetros de entrada de CSC en un amplio abanico de escenarios sintéticos, caracterizados por una alta variabilidad de recursos de cómputo utilizados, tanto por los usuarios locales como por los usuarios de las aplicaciones paralelas. De este modo, el comportamiento de CSC ha sido cuantificado mediante un conjunto de métricas que permiten medir y comparar el rendimiento tanto de las aplicaciones distribuidas, como de las locales, con respecto a las políticas de coscheduling más significativas descritas en la literatura.

La segunda parte de la experimentación ha sido realizada en un laboratorio real de usuarios (entorno NOW de producción), donde se han ejecutado las diferentes cargas distribuidas mientras los estudiantes estaban realizando sus correspondientes prácticas, ya sean dirigidas o bien libres. Estas pruebas han permitido validar la bondad de las métricas definidas en el entorno controlado, así como la correlación de las mismas con respecto a la percepción real tanto de los usuarios de las aplicaciones distribuidas como de las locales.

Sobre ambos entornos se han ejecutado diferentes cargas distribuidas, caracterizadas por diferentes requerimientos, tanto de comunicación, de CPU como de memoria. Cada carga ha sido ejecutada con diferentes grados de multiprogramación (MPL) y con diferentes porcentajes de recursos de cómputo asignados (L), con objeto de poder estudiar los límites de ambos parámetros en un entorno cluster no dedicado.

5.1. Marco de Experimentación

En esta sección se describirán las cargas, tanto distribuidas como locales, utilizadas en esta experimentación, junto con las métricas utilizadas para validar el rendimiento de las mismas.

5.1.1. Caracterización de la carga distribuida

Los benchmarks distribuidos utilizados en la experimentación realizada, tanto en el entorno controlado como en el productivo, se agrupan en dos grandes grupos: *aplicaciones reales,* pertenecientes a la NAS *suite* de Benchmarks [BBB+94], y *aplicaciones sintéticas,* desarrolladas expresamente en esta tesis.

La *NAS Parallel* es una *suite* de aplicaciones paralelas realizadas por la "*Numerical Aerodynamic Simulation (NAS) Program of the National Air and Space Administration (NASA)*" para el análisis del rendimiento de computadores paralelos. Dada la inexistencia de un consenso entre la comunidad científica en cómo deberían de ser los benchmarks paralelos, la NASA realizó los benchmarks en aquellas áreas que más le interesaban: la computación en dinámica de fluidos (CFD). Esta *suite* consta de cinco "kernels", que implementan algoritmos numéricos de uso común en aplicaciones CFD, y tres aplicaciones complejas, denominadas SP, BT y LU, que simulan el comportamiento de aplicaciones CFD completas. Para cada uno de los benchmarks, se definen cuatro clases diferentes (T, A, B y C), las cuales difieren en el tamaño del problema a resolver, siendo la clase T la de menor tamaño y la clase C la de mayor tamaño. La tabla 5.1 muestra los requerimientos medios de comunicación (*Comm*) y de cómputo (*Comp*) expresados en porcentajes de tiempo de ejecución, así como los requerimientos de memoria por nodo (*Mem*) de cada uno de los benchmarks NAS utilizados en esta experimentación. Los porcentajes de comunicación y cómputo son devueltos por cada uno de los benchmarks al final de su ejecución, mientras que los requerimientos de memoria han sido calculados por medio de la herramienta de monitorización, *MoniTo*, descrita en el apéndice D. Cada benchmark ha sido

ejecutado con tres tamaños diferentes (4, 8 y 16 tareas). Los requerimientos asociados a cada uno de los tamaños se muestran en la tabla 5.1. Un estudio detallado acerca del patrón de comunicación de cada uno de los benchmarks utilizados puede ser consultado en [VM02], asimismo en [WMADC99] se realiza un estudio exhaustivo acerca de la escalabilidad de dichos benchmarks en entornos cluster dedicados.

Benchmarks	**%Comp 4/8/16**	**%Comm 4/8/16**	**Mem (MB) 4/8/16**
EP.A	99/99/99	1/1/1	0.6/0.6/0.6
EP.B	99/99/99	1/1/1	0.6/0.6/0.6
IS.A	49/42/28	51/58/72	72/36/23
IS.B	47/39/26	53/61/74	277/117/48
MG.B	82/76/70	18/24/30	120/63/31
FT.A	51/44/40	49/56/60	129/61/26
CG.A*	70/54/48	30/46/52	55/55/55
CG.B*	72/61/52	28/39/48	112/112/112
LU.A	94/88/78	6/12/22	25/25/13
SP.A	85/83/79	15/17/21	44/23/13
BT.B	96/92/87	4/8/13	22/14/8

Tabla 5.1: Requerimientos de CPU (*Comp),* comunicación (*Comm*) y memoria (*Mem*) de los NAS benchmarks. La primera columna muestra el nombre del benchmark junto con la clase con la que que ha sido ejecutado. *El tamaño del benchmark CG ha de ser un cuadrado perfecto (4, 9 ó 16 tareas).

Asimismo, se ha utilizado el benchmark sintético *Sintree*, descrito en la sección 2.3.3. Este benchmark permite fijar al usuario, los requerimientos de comunicación, de CPU y de memoria; de manera que permite modelar distintas cargas con una gran variedad de requerimientos de comunicación y de cómputo. En concreto, este benchmark ha sido ejecutado con una frecuencia de comunicación (*freq*) baja ($= 0{,}5KB/s$), media ($= 5KB/s$) y alta ($= 50KB/s$). Los requerimientos de comunicación, memoria y CPU para cada uno de los tres casos evaluados se muestran en la tabla 5.2.

Los benchmarks anteriormente descritos fueron combinados, de acuerdo con sus requerimientos de comunicación, en tres cargas distribuidas diferentes, formadas cada una de ellas por aquellos benchmarks con altos (COM_H), medios (COM_M) y bajos (COM_L) requerimientos de comunicación, respectivamente. La tabla 5.3 muestra los benchmarks que integran las diferentes agrupaciones realizadas, junto con el porcentaje de comuni-

Benchmarks	**freq (KB/s)**	**%Comp 4/8/16**	**%Comm 4/8/16**	**Mem (MB) 4/8/16**
Sintree-L	0.5	97/97/97	3/3/3	40/40/40
Sintree-M	5	78/76/73	22/24/27	40/40/40
Sintree-H	50	31/29/26	69/71/74	40/40/40

Tabla 5.2: Requerimientos de cómputo (*Comp*), comunicación (*Comm*) y memoria (*Mem*) del benchmark *Sintree* en función de la frecuencia de comunicación (freq).

cación medio obtenido ($\overline{\%Comm}$) para cada una de las cargas definidas, cuando éstas fueron ejecutadas en el cluster de 4, 8 y 16 nodos, respectivamente. Cada carga está formada por 24 benchmarks, escogidos aleatoriamente a partir del conjunto de 6 benchmarks mostrados en la tabla 5.3. Estos 24 benchmarks fueron ejecutados en sucesivas iteraciones, de modo que en cada iteración se ejecutaron MPL trabajos concurrentemente. Los MPL trabajos a ejecutar en cada iteración fueron escogidos en dos fases sucesivas, de acuerdo con el modelo de carga definido por Feitelson [Fei96]: en primer lugar, se escogió si se ejecutaba una trabajo grande (NAS clase B y Sintree), con probabilidad 0,4 o bien un trabajo pequeño (NAS clase A), con probabilidad 0,6, de acuerdo con una distribución de Bernoully, y posteriormente, se seleccionó el trabajo, perteneciente a la clase escogida en la etapa anterior, de acuerdo con una distribución uniforme. Asimismo, cada prueba fue repetida cinco veces, de modo que los resultados mostrados a lo largo de esta experimentación son los valores medios obtenidos a lo largo de las cinco pruebas. En la sección B.3 del apéndice B se muestran y se analizan las desviaciones estándar asociadas a dichas medias.

Cargas Distribuidas	**Benchmarks**	$\overline{\%Comm}$ 4/8/16
COM_L	EP.A, LU.A, SP.A, EP.B, Sintree-L, BT.B	5/7/10
COM_M	LU.A, FT.A, MG.B, SP.A, Sintree-M, BT.B	19/23/29
COM_H	IS.A, IS.B, FT.A, MG.B, Sintree-H, CG.B	45/53/61

Tabla 5.3: Relación de benchmarks que integran las cargas distribuidas generadas.

5.1.2. Caracterización de la carga local

Las tareas locales han sido simuladas mediante un benchmark sintético, denominado *Local*, que permite fijar los requerimientos de memoria, de CPU, de E/S a disco y de comunicación utilizados por un usuario local. Asimismo, con objeto de conocer el grado de intrusión introducido por la ejecución de las aplicaciones distribuidas en las tareas locales, este mismo benchmark retorna el tiempo de ejecución en realizar un determinado cómputo junto con la latencia media en realizar un determinado número de llamadas a sistema. El apéndice E describe en detalle la implementación realizada de este benchmark, así como el grado de correspondencia del mismo con respecto al comportamiento de un usuario local real.

Con objeto de fijar los parámetros de entrada del benchmark *Local*, de un modo realista, se monitorizaron durante dos semanas, las 24 horas del día, 10 puestos de trabajo del laboratorio de usuarios de la *Escuela Politécnica Superior* (EPS) de la *Universitat de Lleida* (UdL). Este laboratorio se caracteriza porque está abierto a los estudiantes, tanto por el día como por la noche, de modo que estos pueden realizar libremente sus respectivos trabajos. Todos los puestos de trabajo monitorizados disponen de la misma configuración, tanto hardware como software. Cada máquina dispone de un Pentium IV a $1{,}8Ghz$ con $256MB$ de memoria principal, gestionado por una distribución *Linux Red-Hat 7.2*. Los resultados obtenidos en dicha monitorización evidencian que los diferentes usuarios locales pueden agruparse bajo tres diferentes perfiles, cuyos requerimientos son mostrados en la tabla 5.4. La columna *Carga_CPU* corresponde al número medio de tareas en la cola de preparados durante un intervalo de un minuto, la columna indicada como *%Memoria* corresponde al porcentaje[1] de memoria principal utilizada por las tareas locales, la columna de *E/S a disco* corresponde al número de bloques por segundo leídos y escritos desde disco, mientras que la última columna, denominada *Red*, corresponde al número de Bytes por segundo, recibidos y enviados por las tareas locales. En paréntesis se indica la desviación estándar asociada a cada valor.

Asimismo, la figura 5.1 muestra como se distribuyen los usuarios de un laboratorio real, de acuerdo con los tres perfiles de usuario anteriormente definidos. Los resultados obtenidos, en la monitorización realizada, reflejan como el usuario *Xwin* es el usuario más común (*62 %*), mientras que un *33 %* de usuarios se corresponden con el perfil de *Internet* y solamente el *5 %* se corresponde con el perfil *Shell*. De este modo, el perfil asociado con cada instancia ejecutada del benchmark *Local* ha sido definido atendiendo a estos

[1]Este valor comprende la memoria utilizada por el s.o. y el entorno de ventanas correspondiente.

Local	*Carga_ CPU*	*%Memoria*	*E/S a disco (blocs/s) Leídos - escritos*	*Red (Bytes/s) Rec. - Env.*
Shell	0.25 (0.2)	20 % (15 %)	13 - 12 (4-3)	108 - 3 (30-1)
Xwin	0.15 (0.1)	35 % (55 %)	67 - 17 (23-12)	608 - 30 (302-5)
Internet	0.2 (0.1)	60 % (75 %)	99 - 52 (54-21)	3154 - 496 (1890-245)

Tabla 5.4: Requerimientos medios de los usuarios locales. En paréntesis se muestran las desviaciones estándar asociadas a cada valor medio.

porcentajes. Asimismo, con el objetivo de simular la alta variabilidad en el comportamiento de un usuario local, el tiempo de ejecución del benchmark local ha sido modelado, de acuerdo con los estudios realizados por *Mutka y Livny* [ML91], mediante una distribución hiper-exponencial de dos etapas, con medias de 20min. y 60min. y con un peso de 0,4 y 0,6, respectivamente.

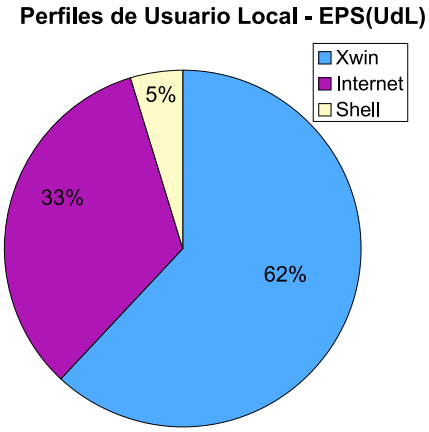

Figura 5.1: Distribución del tipo de usuarios locales.

5.1.3. Métricas de rendimiento

La eficiencia de las políticas de coscheduling, en general, y de CSC, en particular, puede ser analizada bajo tres diferentes ópticas, la percepción del usuario de las aplicaciones paralelas, el cual lógicamente busca ejecutar

sus aplicaciones distribuidas tan rápidas como sea posible, el punto de vista del usuario local, el cual cede su máquina siempre y cuando no perciba una caída de prestaciones y finalmente el punto de vista del administrador del entorno paralelo, donde se ejecutan ambos tipos de aplicaciones, el cual espera ejecutar la carga distribuida lo más rápidamente posible, perturbando mínimamente las aplicaciones del usuario local.

Lógicamente, el usuario de las aplicaciones paralelas desea que aquella aplicación (A_j) que es ejecutada en un entorno monoprocesador dedicado ($node_i$) en un tiempo $T(A_j, node_i)$, sea ejecutada en un tiempo menor ($T(A_j, Cl)$) en el entorno cluster (Cl). Con objeto de cuantificar esta relación, se ha utilizado la métrica speedup medio a nivel de aplicación, denotada como $\overline{SP\_Ap}(Wrk)$. Esta métrica es definida de la siguiente manera:

$$\overline{SP\_Ap}(Wrk) = \sum_{j=1}^{N} \frac{min_{i=1}^{n}\{T(A_j, node_i)\}}{T(A_j, Cl) * N}, \tag{5.1}$$

donde n es el número de nodos del cluster y N ($= 24$) es el número de aplicaciones (A_j) que forman la carga distribuida evaluada (Wrk). Cabe decir, que el numerador de la expresión 5.1 refleja un entorno heterogéneo, donde las máquinas que forman el cluster ofrecen diferentes prestaciones y por tanto, para el cálculo del speedup, se toma el tiempo obtenido en la máquina más rápida.

El speedup a nivel de aplicación, definido en la expresión 5.1, no es representativo del rendimiento global del sistema con respecto a la ejecución de toda la carga distribuida generada (Wrk) y por tanto, no es útil para el administrador del entorno NOW. Con objeto de obtener una métrica asociada a la ejecución de toda la carga distribuida, se ha calculado el speedup de la carga distribuida ($SP(Wrk)$), de acuerdo con la siguiente expresión:

$$SP(Wrk) = \frac{min_{i=1}^{n}\{T(Wrk, node_i)\}}{T(Wrk, Cl)}, \tag{5.2}$$

donde $T(Wrk, Cl)$ es el tiempo de ejecución de toda la carga distribuida (Wrk) en el entorno cluster (Cl), es decir, el tiempo transcurrido desde el inicio de la ejecución del primer benchmark asociado a Wrk, hasta que finaliza la ejecución del último benchmark asociado a dicha carga. El numerador de la expresión 5.2 ($min_{i=1}^{n}\{T(Wrk, node_i)\}$) es el tiempo obtenido al ejecutar la carga Wrk secuencialmente en el nodo más rápido del cluster. Este tiempo es calculado de acuerdo con la siguiente expresión:

$$T(Wrk, node_i) = \sum_{j=1}^{N} T(A_j, node_i), \tag{5.3}$$

siendo N el número de aplicaciones (A_j) que forman la carga Wrk. La diferencia entre ambas métricas ($SP(Wrk)$ y $\overline{SP\_Ap}(Wrk)$) es sustancial cuando se incrementa el grado de multiprogramación paralela, dado que puede ocurrir que mientras el speedup a nivel de aplicación sea relativamente bajo, el speedup asociado a la ejecución de toda la carga distribuida sea alto, como consecuencia de un uso más eficiente de los recursos del sistema, debido al incremento del grado de multiprogramación paralela.

Con respecto al usuario local, el benchmark sintético implementado (*Local*) retorna dos diferentes métricas. Una descripción detallada de ambas métricas puede ser encontrada en el apéndice E. La primera de ellas, la *latencia media de las llamadas a sistema* (LSC : "*Latency System Call*"), corresponde a un promedio realizado a partir de la latencia obtenida en la ejecución de 20 llamadas a sistema diferentes, representativas del trabajo de un usuario interactivo; como por ejemplo son las llamadas para la creación y eliminación de procesos, creación y eliminación de ficheros, lecturas y escrituras, tanto a disco como a memoria, etc... De acuerdo con los razonamientos expuestos en la sección 3.4, esta métrica debe ser siempre inferior a $400ms$, con objeto de que el usuario local no perciba una ralentización excesiva en el tiempo de respuesta de sus llamadas. La segunda métrica, que retorna este benchmark, es el tiempo medio que tarda en ejecutar un determinado cómputo (algoritmo de *Sieve Eratosthenes* [Jai91]). A partir de este tiempo, se ha calculado el slowdown de acuerdo con la siguiente expresión:

$$Slowdown(Local) = \sum_{i=1}^{k} \frac{T(Local, node_i^{no\_dedicado})}{k * T(Local, node_i^{dedicado})}, \tag{5.4}$$

siendo k el número de instancias del benchmark *Local* ejecutadas en el cluster y $T(Local, node_i^{dedicado/no\_dedicado})$, el tiempo de ejecución que retorna el benchmark *Local* en el $node_i$, tanto cuando el nodo está dedicado o no dedicado al usuario local. El valor máximo de esta métrica, aceptado por el usuario local, ha sido fijado en 2, considerando que un valor superior a dos causaría un retardo inaceptable para el usuario local de aquellas aplicaciones que requieran un cómputo intensivo, durante intervalos de tiempo del orden de minutos. De acuerdo con los resultados obtenidos en la monitorización de las aulas de laboratorios de usuarios, descrita en la sección anterior, este límite de dos puede ser considerado como conservador, dado que los usuarios locales caracterizados por un alto uso de la CPU (5 % del total) suelen ejecutar aplicaciones de cómputo intensivo del orden de segundos.

5.2. Evaluación de CSC en un Entorno NOW Controlado

El entorno NOW controlado se corresponde con un cluster cerrado al público, de manera que toda la carga que se ejecuta sobre él está controlada. Este cluster, denominado *Cl_ Control*, está formado por 16 PCs heterogéneos, los cuales se agrupan, atendiendo a las prestaciones de los PCs, en dos subclusters diferentes totalmente homogéneos, denominados *Cl1_ Control* y *Cl2_ Control*, respectivamente. Cada PC de *Cl1_ Control* dispone de un procesador PentiumII a 350mhz, con 128MB de RAM, 128MB de memoria Swap y un disco de 3GB, mientras que los PCs pertenecientes a *Cl2_ Control* disponen de procesadores PentiumIII a 800Mhz, con 256MB de memoria RAM, 256MB de memoria de Swap y un disco de 30GB. Los 16 PCs del cluster disponen de la misma configuración, una distribución Red-Hat 6.2 con un kernel v.2.2.15 y el entorno de paso de mensajes PVM 3.4. La red de interconexión es una Fast Ethernet a 100Mbps con una latencia mínima de $0{,}1 msg$.

La tabla 5.5 muestra, a modo de referencia, los tiempos de ejecución obtenidos al ser ejecutado cada uno de los NAS benchmarks, así como los benchmarks sintéticos distribuidos, en un entorno dedicado a los mismos; es decir sin carga local y con un grado de multiprogramación igual a uno. Cada benchmark ha sido ejecutado con diferente número de tareas: 1, 4, 8 y 16 tareas. Las ejecuciones con 4 y 8 tareas han sido realizadas tanto en el *subcluster Cl1_ Control* como en el *Cl2_ Control*, mientras que la ejecución monoprocesador (1 tarea) ha sido realizada en un PC del subcluster con mayores prestaciones (*Cl2_ Control*).

En todas las pruebas realizadas en este entorno se ha evaluado el rendimiento de CSC con respecto al coscheduling Dinámico tradicional (*Dynamic*) y a un esquema de planificación no coordinado, como es el s.o. *Linux* original. La eficiencia de las tres políticas evaluadas ha sido comparada tanto en lo que respecta al rendimiento de las tareas locales como al rendimiento de las tareas distribuidas. Atendiendo a los resultados mostrados en la sección B.2 del apéndice B, los valores de los principales parámetros de CSC son:

- *STEP quantum.* De acuerdo con la definición dada en la sección 3.4.1, el quantum de tiempo asignado a todos los procesos distribuidos residentes en aquellos nodos, donde la memoria ha sido desbordada y no existe actividad de usuario local, es incrementado, con respecto a la longitud de quantum asignado por defecto por el s.o, en un factor constante denominado *STEP*. Este parámetro, atendiendo a dicha experimentación, ha sido asignado igual a 5.

Benchmarks	Tiempo Ejec.(s)					
	Cl1_Control		Cl2_Control			Cl_Control
	4	8	1	4	8	16
EP.A	270	135	489	118	59	67
EP.B	1083	536	1904	473	236	241
IS.A	101	84	146	87	73	71
IS.B	530	172	1844	323	154	146
MG.B	482	298	1980	278	200	195
FT.A	632	253	797	410	208	194
CG.B	465	405	1805	395	385	374
LU.A	200	123	557	110	86	65
SP.A	289	168	752	236	143	142
BT.B	574	351	932	309	237	251
Sintree-L	594	298	1520	512	246	158
Sintree-M	395	202	968	389	198	141
Sintree-H	251	146	786	243	142	101

Tabla 5.5: Tabla de tiempos de ejecución de los NAS benchmarks y benchmarks sintéticos en el cluster dedicado controlado y con un grado de multiprogramación igual a uno.

- MEM_MIN. Este parámetro, definido en la sección 3.4, se corresponde al porcentaje máximo de la memoria utilizada que puede tener un nodo para que CSC reanude aquella tarea distribuida previamente detenida (*stoptask*) debido a una situación de excesiva paginación. De acuerdo con la experimentación mostrada en la sección B.2 del apéndice B, este parámetro ha sido fijado igual al 95 %.

5.2.1. Porcentaje de recursos de cómputo (L) asignado a las tareas distribuidas

En esta sección se procede a evaluar el rendimiento, tanto de las tareas locales como de las distribuidas, en función del porcentaje de recursos asignados a las tareas distribuidas (L), definido en las secciones 3.2 y 3.4 del presente documento. Con respecto a la CPU, este porcentaje L significa que aquellas tareas distribuidas residentes en nodos con actividad de usuario local, así como el resto de tareas, residentes en nodos remotos sin actividad local, que forman parte de la misma aplicación, tendrán asignada una rebanada de tiempo igual a $DEF\_QUANTUM * L$, siendo $DEF\_QUANTUM$ el

valor del quantum asignado, por defecto, a todas las tareas locales y a aquellas tareas distribuidas no perturbadas por el usuario local, ya sea directa o indirectamente. Asimismo, con respecto a la memoria, el porcentaje L limita el tamaño máximo de memoria que puede ser utilizado por las tareas distribuidas. Con el objetivo de aislar la influencia de dicho porcentaje del grado de multiprogramación paralelo (MPL), el parámetro MPL ha sido fijado en tres a lo largo de todas las pruebas mostradas en esta sección. Toda la experimentación descrita en esta sección ha sido realizada en el subcluster homogéneo de 8 máquinas, *Cl2_ Control*.

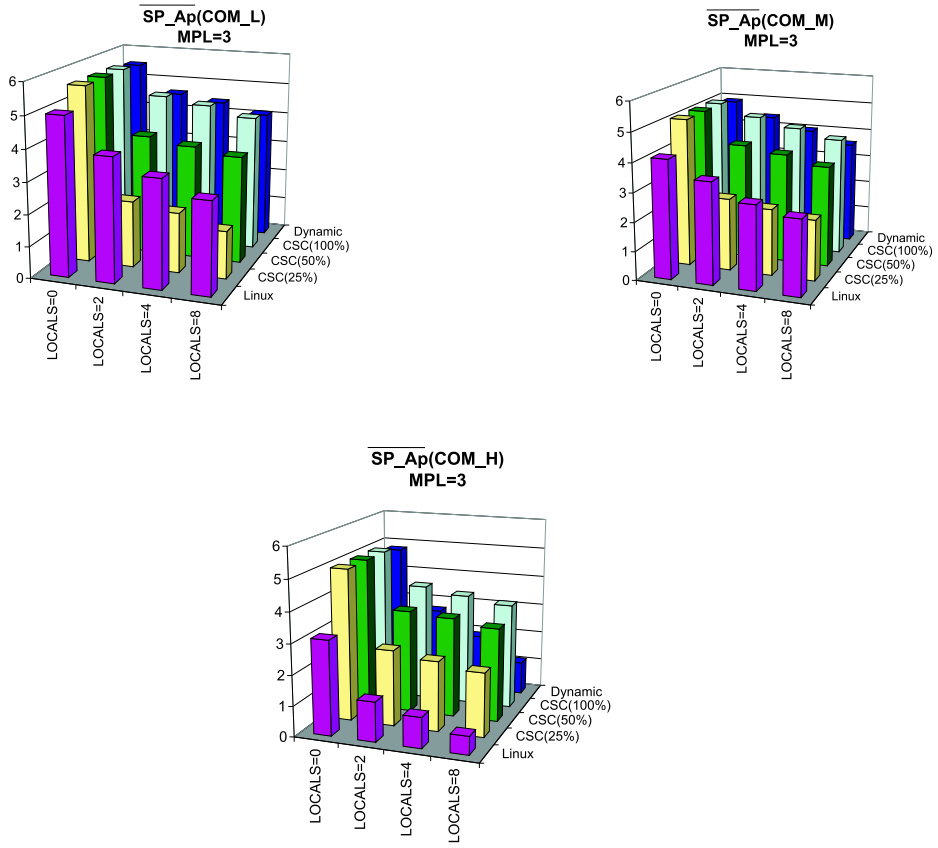

Figura 5.2: Speedup medio de las aplicaciones distribuidas en función del porcentaje de recursos asignado (L) a las mismas y del número de usuarios locales (LOCALS).

La figura 5.2 muestra el speedup medio de las aplicaciones que integran las tres cargas distribuidas definidas, *COM_L*, *COM_M* y *COM_H*, en función de tres diferentes porcentajes de recursos (*L= 25, 50 y 100 %*) y

para diferente número de usuarios locales presentes a lo largo del cluster ($LOCALS$=0, 2, 4 y 8); correspondiendose $LOCALS = 0$ al caso donde el cluster esté totalmente dedicado a las aplicaciones paralelas y $LOCALS = 8$ al caso donde en todos los nodos reside un usuario local. Asimismo, en esta figura, se muestran, a modo de referencia, los resultados obtenidos con Linux y el coscheduling Dinámico.

En general, los resultados mostrados en la figura 5.2 evidencian una alta dependencia con respecto al tipo de carga distribuida. Los valores obtenidos con la carga caracterizada por tener bajos requerimientos de comunicación (COM_L) muestran como el rendimiento de CSC decae proporcionalmente al valor de L. No obstante, es importante remarcar que aun en el peor de los casos para las tareas distribuidas (L=*25 %*), el speedup siempre es superior a uno, lo cual indica que el usuario paralelo continua ganando con respecto al caso de la ejecución secuencial. Asimismo, como cabía esperar, los mejores resultados se obtienen con el coscheduling Dinámico y con CSC con L=100 %, aunque los resultados obtenidos bajo Linux no son, en absoluto, despreciables. Este buen resultado de Linux es debido a la escasa necesidad de coscheduling que tienen los trabajos distribuidos que forman esta carga. Sin embargo, a medida que aumentan las necesidades de comunicación de las tareas distribuidas, el rendimiento de Linux con respecto al resto de políticas evaluadas decrece hasta alcanzar la situación extrema, con la carga COM_H y con todo el cluster utilizado por usuarios locales ($LOCALS$=*8*), donde el speedup obtenido por Linux está por debajo de uno. Asimismo, el análisis de las cargas con comunicación media y alta muestra, como a medida que aumentan las necesidades de coscheduling de las tareas distribuidas, la diferencia entre los valores obtenidos con un L=100 % y un L=50 % se reduce considerablemente. Este comportamiento es debido a que estas tareas se caracterizan por tener de forma reiterada cortos periodos de cómputo, de manera que no suelen consumir el quantum asignado por defecto por el s.o. (210ms), con lo que si CSC(50 %) reduce este quantum a la mitad (L=50 %), su rendimiento se ve escasamente afectado. No obstante, en el caso de que el quantum sea disminuido al 25 % del original, el rendimiento de las tareas distribuidas se resiente de manera apreciable. Asimismo, comparando los resultados obtenidos con CSC(100 %) y con el coscheduling Dinámico se observa como la diferencia aumenta a medida que se incrementa la tasa de comunicación de las aplicaciones distribuidas, así como el grado de multiprogramación. Esta mejora de CSC(100 %) es atribuible al efecto del algoritmo de coscheduling, implementado en el mismo, que es capaz de discernir entre tareas distribuidas con similar frecuencias de comunicación y que pertenecen a distintas aplicaciones, hecho que repercute en una mejora importante en la coplanificacón de tareas remotas cooperantes,

y por consiguiente en una mejora de los tiempos de ejecución de las aplicaciones distribuidas. Finalmente, en contra de los esperado, caben destacar los pobres resultados obtenidos por el coscheduling Dinámico con la carga COM_H. Este pobre resultado se debe a que esta carga y para este grado de multiprogramación ($MPL = 3$) provoca, en aquellos nodos donde hay usuarios locales, que la memoria sea desbordada, hecho que provoca, como ya se justificó en la simulación realizada, que la probabilidad de alcanzar el coscheduling sea escasa. A este respecto, cabe destacar como CSC, debido a su capacidad de analizar los requerimientos de memoria y adaptar el grado de multiprogramación en consecuencia, sea capaz de mantener el speedup de las tareas distribuidas prácticamente constante, independientemente del número de usuarios locales en el cluster. En la sección B.2 del apéndice B se muestra un análisis detallado del rendimiento de CSC en aquellos casos particulares donde la memoria principal es desbordada, ya sea debido a que los requerimientos del usuario local sobrepasan su porción asignada o bien a que las tareas distribuidas superan dicha porción.

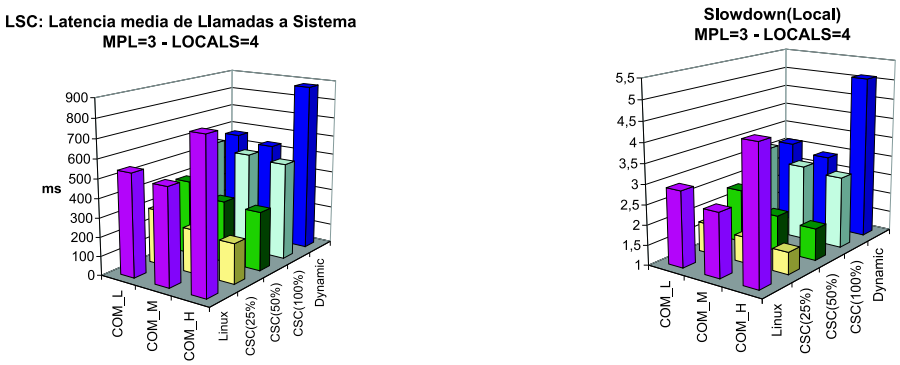

Figura 5.3: Latencia media de las llamadas a sistema (izquierda) y slowdown de las tareas locales (derecha).

Con objeto de medir la influencia del porcentaje L en el rendimiento del usuario local se calcularon, bajo las mismas condiciones anteriores, la latencia media de las llamadas a sistema (LSC) y el slowdown de las tareas locales. La figura 5.3 muestra los resultados medios obtenidos para el caso de cuatro usuarios locales presentes en el cluster (LOCALS=4). Tanto Linux, Dynamic, como CSC con L=100 % obtienen unos resultados inadmisibles para el usuario local, dado que tanto la latencia de las llamadas a sistema es superior al umbral fijado de $400ms$, como el slowdown es superior al umbral máximo de 2. Cabe destacar que los picos obtenidos por Linux y Dynamic

con la carga *COM_H* son debidos al desbordamiento de la memoria, y al consiguiente trasiego de páginas con el área de swap. Estos resultados reflejan la necesidad de que una política de coscheduling orientada a entornos no dedicados ha de ser capaz de reasignar los recursos de cómputo de las tareas distribuidas acorde con los requerimientos del usuario local. Esta característica se refleja en el rendimiento alcanzado por CSC para los casos evaluados con L igual al 50 % y al 25 %. En estos casos, ambas métricas de usuario local disminuyen drásticamente, conservando el rendimiento de los usuarios locales por debajo de los umbrales máximos permitidos. En este sentido cabe destacar como CSC es capaz de mantener la eficiencia del usuario local en aquellos casos donde existe una gran demanda de requerimientos de memoria por parte de las tareas distribuidas, como es el caso de la carga *COM_H*.

En conclusión, los resultados mostrados en esta sección revelan la necesidad de limitar los recursos utilizados por las tareas distribuidas, siendo el porcentaje del 50 % el que mejor se ajusta a las necesidades tanto del usuario local como del usuario de las aplicaciones distribuidas. En este sentido, cabe destacar como CSC es capaz de mantener un equilibrio entre el rendimiento de ambos tipos de usuarios, de modo que aun bajando los recursos asignados a las tareas distribuidas obtiene un buen rendimiento para las mismas y, al mismo tiempo, conserva las métricas del usuario local por debajo de los umbrales fijados. Un aspecto interesante a remarcar con respecto al rendimiento de las aplicaciones distribuidas en los entornos cluster no dedicados es que el speedup obtenido en la ejecución de las mismas no supone coste económico alguno, desde el punto de vista de inversión hardware, para la empresa o institución propietaria del cluster.

De acuerdo con los resultados presentados a lo largo de esta sección, el resto de la experimentación mostrada a lo largo de este capítulo se ha realizado aplicando un porcentaje *L=50 %* para la política CSC.

5.2.2. Grado de multiprogramación de las tareas distribuidas

En esta sección se evaluará el comportamiento de las tres políticas anteriormente descritas, CSC con L=50 %, Linux y el coscheduling Dinámico (Dynamic), con respecto al grado de multiprogramación paralela (MPL). Estas pruebas han sido realizadas en el subcluster homogéneo de 8 máquinas, *Cl2_Control*, con la mitad de los nodos ocupados por usuarios locales (*LOCALS=4*). Este número de usuarios locales no ha sido escogido al azar, si no que se ha tomado atendiendo tanto a los índices de ocupación que presentan los laboratorios de usuarios de nuestra escuela (*Escuela Politécnica Superior*

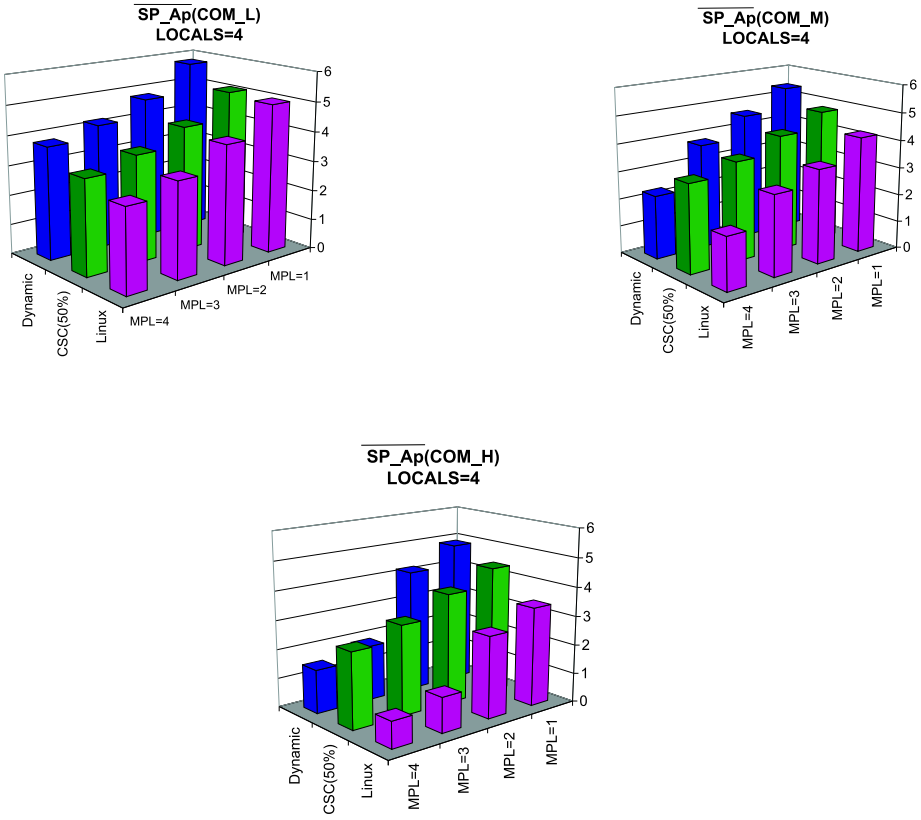

Figura 5.4: Speedup de las aplicaciones distribuidas en función del grado de multiprogramación (MPL).

de la UdL), así como a las ratios de ocupación mostradas en diferentes estudios de la literatura [AES97, ML91]. Dichos estudios reflejan que la mayor parte del tiempo, más de la mitad de las máquinas están desocupadas.

La figura 5.4 muestra el speedup medio obtenido para las aplicaciones que integran cada una de las cargas distribuidas definidas. Con respecto a la carga con altos requerimientos de cómputo (COM_L) se observa como el coscheduling Dinámico obtiene los mejores resultados, mientras que CSC obtiene mejores resultados que Linux; ganancia que se incrementa a medida que se aumenta el grado de multiprogramación. Asimismo, esta diferencia entre CSC y Linux se incrementa a medida que se aumentan los requerimientos de comunicación de las tareas (COM_M y COM_H), como consecuencia del incremento de las necesidades de coscheduling de las tareas distribuidas. Asimismo, los resultados obtenidos reflejan como el rendimiento de Dynamic

cae drásticamente para la carga *COM_M* con MPL=4 y la carga COM_H con un MPL=3 y 4. Esta caída es debida a que Dynamic no tiene en cuenta los requerimientos de memoria de las tareas, hecho que provoca una pérdida de la capacidad de alcanzar el coscheduling en aquellos casos donde se produce un desbordamiento de la memoria. Asimismo cabe decir que, en general, la diferencia entre Dynamic y CSC decrece a medida que se aumenta el grado de multiprogramación MPL, debido a que al aumentar el número de tareas competidoras, las técnicas tradicionales de coscheduling basadas exclusivamente en el análisis de los eventos de comunicación no son capaces, en determinadas situaciones, de mantener la coordinación entre aquellas tareas que se comunican entre si. Entre estas situaciones caben destacar aquellos casos donde las tareas distribuidas que compiten entre si tienen igual frecuencia y patrón de comunicación, hecho que comporta que las políticas de coscheduling tradicionales no sean capaces de discernir entre tareas pertenecientes a distintas aplicaciones. Estos resultados reflejan como CSC obtiene los mejores resultados con aquellas cargas con un volumen de comunicación medio (*COM_M*) o alto (COM_H), debido a su capacidad de coordinar los recursos asignados a las tareas distribuidas a lo largo de todo el cluster.

Comparando el rendimiento de CSC en el entorno cluster con respecto al obtenido en la simulación, presentada en la sección 3.6, se observa como en el entorno cluster controlado CSC obtiene mejores resultados con respecto al resto de políticas evaluadas debido a que en la simulación realizada todas las tareas distribuidas, pertenecientes a un mismo trabajo, tenían los mismos requerimientos de memoria, comunicación y CPU, mientras que en el entorno controlado, los requerimientos de las tareas distribuidas dependen, en gran medida, del flujo de ejecución de las mismas, hecho que favorece a la política CSC debido a su capacidad de asignar dinámicamente los recursos de cómputo de cada nodo atendiendo al estado global de la aplicación a lo largo del cluster.

Con objeto de calcular el rendimiento global de la máquina virtual paralela se procedió a calcular el speedup de las cargas distribuidas. La figura 5.5 muestra el speedup obtenido para las tres cargas evaluadas y para distintos grados de multiprogramación. En primer lugar, cabe resaltar como el speedup obtenido en la mayoría de los casos es superior a 4, siendo este valor igual a la mitad del número de nodos del sistema ($SP(Wrk) > n/2$), umbral considerado por algunos autores [FRS+97] como el speedup mínimo a conseguir para una carga distribuida de n tareas. Asimismo, estos resultados muestran como a medida que se incrementa el grado de multiprogramación, el speedup se va incrementando progresivamente en todos los casos evaluados donde las tareas encajan en memoria principal. En aquellos casos donde los requerimientos de memoria superan el límite de la memoria principal y no

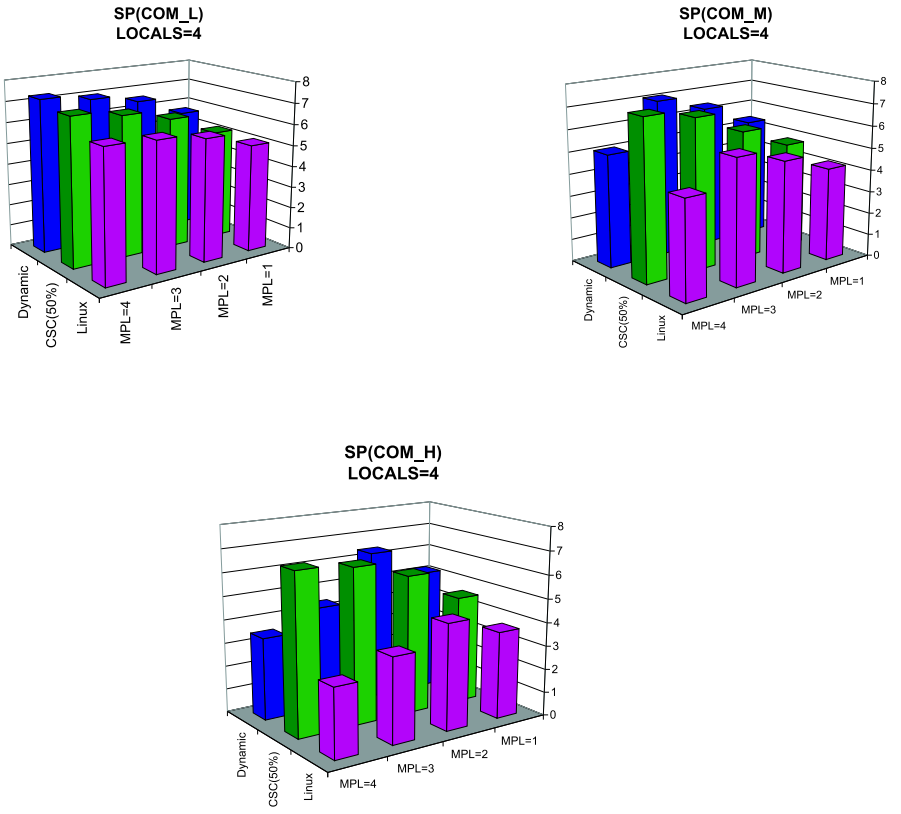

Figura 5.5: Speedup de las cargas distribuidas en función del grado de multiprogramación (MPL).

existe una política de coscheduling que tenga en cuenta los eventos de memoria, caso de Linux y Dynamic, esta evolución del speedup se invierte hasta alcanzar unos valores inferiores a 4. Particularizando en el caso de CSC, los resultados obtenidos reflejan como, a medida que se incrementa el MPL, la mejora del rendimiento del sistema aumenta ostensiblemente para aquellas cargas con elevada tasa de comunicación, reafirmando las tendencias observadas en el análisis del speedup a nivel de aplicación. Cabe destacar que, en este último caso, la caída en el rendimiento de CSC para $MPL > 3$ es debida a que los requerimientos de memoria de las tareas distribuidas sobrepasan la porción asignada a los mismo, hecho que provoca un desbordamiento de memoria y, como consecuencia, que CSC adapte dinámicamente el grado de multiprogramación a la porción de memoria disponible.

La figura 5.6 muestra la latencia media de las llamadas a sistema (arriba)

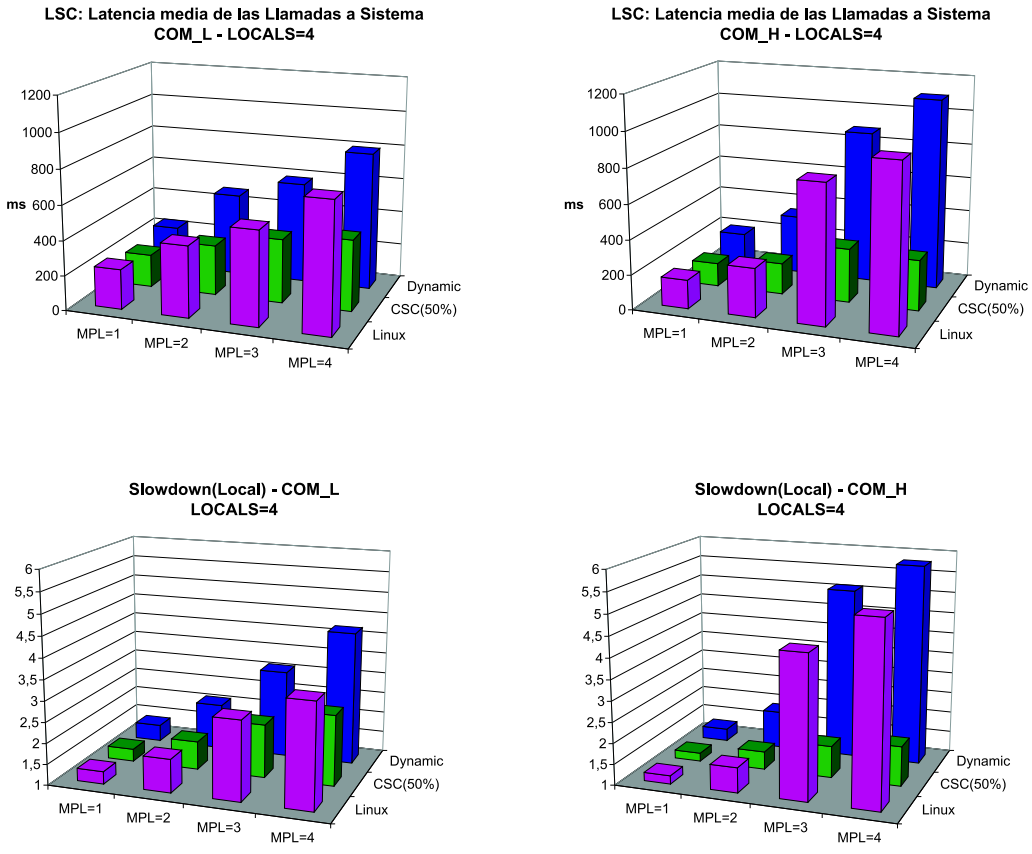

Figura 5.6: Latencia media (arriba) y slowdown (abajo) de las tareas locales en función del grado de multiprogramación paralelo.

y el slowdown de las tareas locales (abajo), cuando éstas son ejecutadas junto con las dos cargas distribuidas extremas: COM_L con altos requerimientos de cómputo y COM_H con altos requerimientos de comunicación. En general, se observa como en todos los casos estudiados, ambas métricas aumentan gradualmente a medida que se aumenta el MPL, siendo CSC el que mejor se ajusta, con diferencia, a los requerimientos del usuario local. Entre estos resultados destacan los picos obtenidos, tanto por Linux como por Dynamic, para la carga COM_H con un $MPL \geq 3$. Estos súbitos incrementos, tanto de la latencia como del slowdown, son debidos a que la memoria de aquellos nodos con usuarios locales es desbordada, de modo que el s.o. local penaliza tanto a las tareas locales como a las distribuidas debido a que no tiene capacidad de discriminar entre las mismas. Sin embargo, la capacidad de CSC de detectar la presencia de los usuarios locales, le permite mantener

el rendimiento de las tareas locales dentro de unos límites aceptables, penalizando a la tarea distribuida con mayor requerimientos de memoria. Por este motivo, CSC mantiene, en estos casos particulares, tanto la latencia como el slowdown prácticamente constantes para $MPL \geq 2$. Con respecto a la incidencia del coscheduling Dinámico en el trabajo del usuario local, cabe destacar como los resultados obtenidos en el entorno cluster son peores que los que se obtuvieron por medio de la simulación (ver sección 3.6). Esta diferencia es debida principalmente a que la condición de apropiación de la CPU implementada en el simulador es mucho más restrictiva, con respecto a las tareas distribuidas, que el mecanismo utilizado por el s.o. Linux para preservar el rendimiento de las tarea interactivas.

En conclusión, los resultados obtenidos en esta sección revelan que en aquellos entornos que no incorporan medidas de limitación de los recursos utilizados por las tareas paralelas, caso de Dynamic y Linux, el grado de multiprogramación de las mismas no puede ser superior a uno para cargas con elevados requerimientos de cómputo y a dos, siempre y cuando la memoria no sea desbordada, para cargas con elevados requerimientos de comunicación. Estas restricciones son debidas a la alta penalización que introducen estas políticas en el rendimiento del usuario local. En cambio, CSC permite doblar el grado de multiprogramación paralela utilizado debido tanto a su capacidad de coordinar la ejecución de las aplicaciones paralelas a lo largo del cluster, como de limitar el overhead causado al usuario local. En concreto, los resultados obtenidos por CSC reflejan que para cargas con altos requerimientos de cómputo, el grado de multiprogramación no puede ser superior a dos, básicamente debido a que el slowdown del usuario local puede superar el umbral máximo fijado de dos, aunque no así la latencia de las llamadas a sistema. Sin embargo, para cargas con elevados requerimientos de comunicación, el grado de multiprogramación puede ser ampliado hasta un máximo de cuatro. Este incremento en el grado de multiprogramación paralela se traduce en una mejora global del speedup a nivel de sistema obtenido por CSC. Estos resultados extienden las conclusiones obtenidas en la sección anterior, en el sentido que CSC obtiene sus mejores resultados globales para cargas con altos requerimientos de comunicación.

5.2.3. Rendimiento de CSC en un cluster heterogéneo

En esta sección se evalúa la capacidad de CSC de mantener la coordinación entre tareas cooperantes residentes en nodos heterogéneos. Con este fin, las cargas *COM_L* y *COM_H* fueron ejecutadas en el cluster de 16 nodos (*Cl_Control*), compuesto por los dos subclusters de 8 nodos, *Cl1_Control* y *Cl2_Control*. Con objeto de caracterizar la potencia relativa de cada $node_i$

con respecto al procesador más rápido presente en el cluster ($node_{max}$), se utilizó la métrica *Power Weight* ($node_i.W$) definida en la sección 3.4.2. Cada nodo de *Cl1_ Control* está caracterizado por un *Power Weight* igual a 0,33 y un tamaño de memoria principal de $128MB$, mientras que cada nodo de *Cl2_ Control* dispone de un *Power Weight* igual a 1 y un tamaño de memoria principal de $256MB$.

Con objeto de balancear la carga distribuida asignada a cada subcluster de acuerdo con las prestaciones del mismo, el grado de multiprogramación de cada $node_i$ ($node_i.MPL$) fue asignado proporcionalmente a su *Power Weight* de acuerdo con la expresión 3.13 ($node_i.MPL = node_{max}.MPL * node_i.W$) descrita en la sección 3.4.2. De este modo, la asignación de los trabajos a cada subcluster se realizó como ilustra la figura 5.7.

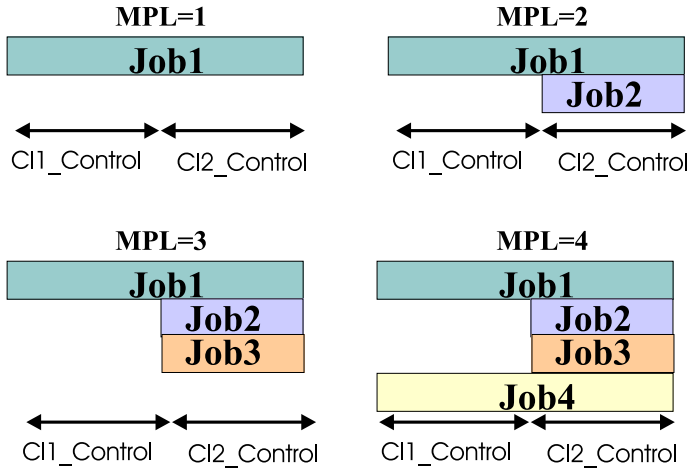

Figura 5.7: Asignación de trabajos en el cluster heterogéneo (*Cl_ Control*).

Un aspecto que debería ser reflejado en este estudio es el comportamiento de los usuarios locales de una aula de informática. Este comportamiento se muestra en el estudio realizado, a raíz de la caracterización de los requerimientos de un usuario local (ver apéndice E), sobre el grado de utilización de los recursos de cómputo de dos aulas de laboratorios con distintas prestaciones, $L1$ y $L3$, ubicadas en la *EPS* de la *UdL*, siendo $L1$ el laboratorio con mayores prestaciones. Esta monitorización fue realizada durante dos semanas consecutivas y durante aquellas horas en que los laboratorios estaban abiertos a los estudiantes para que realizaran sus trabajos sin la supervisión de ningún profesor. En la figura 5.8 se muestra el porcentaje de utilización de cada laboratorio con respecto al número total de usuarios. Estos resultados reflejan como la gran mayoría de usuarios acceden, siempre que hay algún puesto de

trabajo disponible, al laboratorio de mayores prestaciones ($L1$). Con objeto de simular este comportamiento en nuestro cluster controlado, cada vez que se generó un nuevo usuario local, éste fue asignado al cluster *Cl2_ Control* con una probabilidad del 0,75 y al *Cl1_ Control* con una probabilidad del 0,25, de acuerdo con una distribución de *Bernoulli*.

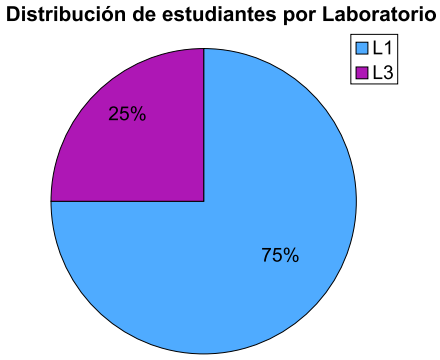

Figura 5.8: Porcentaje de utilización de los laboratorios.

Una cuestión abierta en la literatura científica [SVS02, Kwo03] es la eficiencia que tiene un cluster heterogéneo con respecto a un cluster homogéneo de menor tamaño. Con objeto de comparar ambas alternativas para el caso particular de nuestro cluster se obtuvo el speedup correspondiente a cada carga distribuida ejecutada bajo la política CSC, tanto en el cluster heterogéneo de 16 nodos *(Cl_ Control)* como en el subcluster homogéneo de 8 nodos *(Cl2_ Control)*. Los resultados fueron realizados considerando un cluster no dedicado, donde la mitad de las máquinas eran compartidas con un usuario local. La tabla 5.6 muestra el speedup asociado a cada una de las cargas bajo estudio para los cuatro grados de multiprogramación evaluados. Estos resultados demuestran como, para el caso concreto del cluster bajo estudio, sale a cuenta para el usuario distribuido, el uso de todo el cluster de 16 nodos, siempre y cuando el grado de multiprogramación sea inferior a cuatro. El mal resultado obtenido con el cluster *Cl_ Control* para el caso $MPL = 4$ es debido al total desbalanceo de la carga asignada a dicho cluster, para este caso particular. Cabe remarcar que el propósito de la experimentación presentada en esta sección no es estudiar el rendimiento de un cluster con respecto a su grado de heterogeneidad, estudio que comporta una elevada complejidad de acuerdo con los numerosos parámetros que intervienen en el mismo, si no demostrar que en aquellos casos particulares donde el rendimiento del cluster

heterogéneo sea satisfactorio, CSC resulta una buena alternativa para mantener la coordinación de las tareas cooperantes atendiendo a la heterogeneidad de los recursos de cómputo.

MPL	SP(COM_L)		SP(COM_H)	
	Cl2_Control	Cl_Control	Cl2_Control	Cl_Control
1	5.3	5.6	4.6	5.1
2	6.3	6.5	5.9	6.3
3	6.8	6.9	6.6	6.8
4	7.1	7	6.8	6.5

Tabla 5.6: Speedup de las cargas distribuidas, *COM_L* y *COM_H,* en el cluster heterogéneo*, de 16 nodos, Cl_Control* y en el cluster homogéneo, de 8 nodos, *Cl2_Control.*

La figura 5.9 muestra el speedup medio de las aplicaciones distribuidas asociadas con las dos cargas extremas, *COM_L* y *COM_H*, al ser ejecutadas, cada una de ellas, en el cluster heterogéneo dedicado (ver figura 5.9(arriba)), es decir, sin usuarios locales, y en el cluster no dedicado con 8 usuarios locales ($LOCALS = 8$) trabajando a lo largo del cluster (ver figura 5.9(abajo)). Para cada entorno se han obtenido los resultados aplicando dos diferentes estrategias de coscheduling: CSC y coscheduling Dinámico (Dynamic).

Los resultados obtenidos en el entorno dedicado (ver figura 5.9(arriba)) revelan como las aplicaciones distribuidas que integran la carga *COM_L,* progresan de un modo similar en las dos estrategias aplicadas, mientras que para la carga *COM_H,* el speedup obtenido con la política CSC resulta sensiblemente mejor. Cabe destacar que la caída en el rendimiento experimentado, para ambas cargas y ambas políticas, para el caso de $MPL = 4$, es debida al total desbalanceo de la carga. Asimismo, para ambas cargas se puede apreciar como el speedup decae lentamente a medida que se incrementa el grado de multiprogramación hasta $MPL = 2$. Este comportamiento es debido al particular mapping de los trabajos en el cluster (ver figura 5.7), el cual provoca que el trabajo que evoluciona a lo largo de todo el cluster progrese al ritmo marcado por el subcluster más lento (*Cl1_Cluster*), de modo que el resto de trabajos que se ejecutan solamente en el subcluster con mayores prestaciones (*Cl2_Cluster*), pueden disponer de la mayoría de recursos de cómputo que ofrece este cluster, con lo cual estos trabajos puede progresar como si estuviesen ejecutándose con un grado de multiprogramación menor.

Sin embargo, los resultados obtenidos en el entorno no dedicado difieren sustancialmente en función de la estrategia aplicada. En este segundo caso

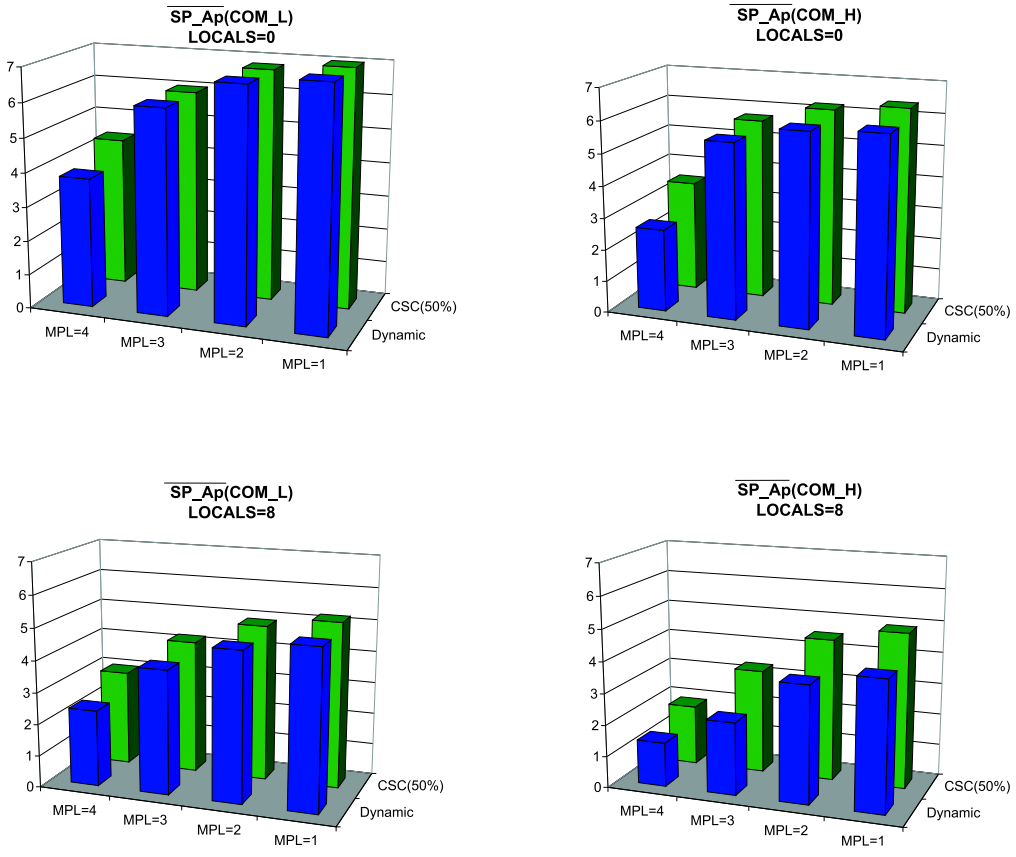

Figura 5.9: Speedup de las aplicaciones distribuidas *COM_L* y *COM_H* en el cluster heterogéneo dedicado (arriba) y no dedicado (abajo).

se observa como los resultados obtenidos con CSC, y especialmente para la carga *COM_H*, son ostensiblemente mejores. La mejora obtenida para la carga *COM_L* es debida a que CSC incrementa el quantum de las tareas distribuidas en función del *Power Weight* de cada máquina, hecho que comporta que la longitud de la época sea incrementada y, por consiguiente, las tareas locales no sean planificadas con tanta frecuencia. De este modo, durante cada época de planificación, todas las tareas que integran una misma aplicación progresan coordinadamente a lo largo del cluster, independientemente del *Power Weight* de la máquina en que residen. Cabe decir que este efecto aumenta con el grado de multiprogramación y los requerimientos de comunicación de las tareas distribuidas, hecho que explica la importante ganancia obtenida por CSC para el caso de la carga *COM_H*.

En contrapartida, este incremento de la longitud del quantum por parte

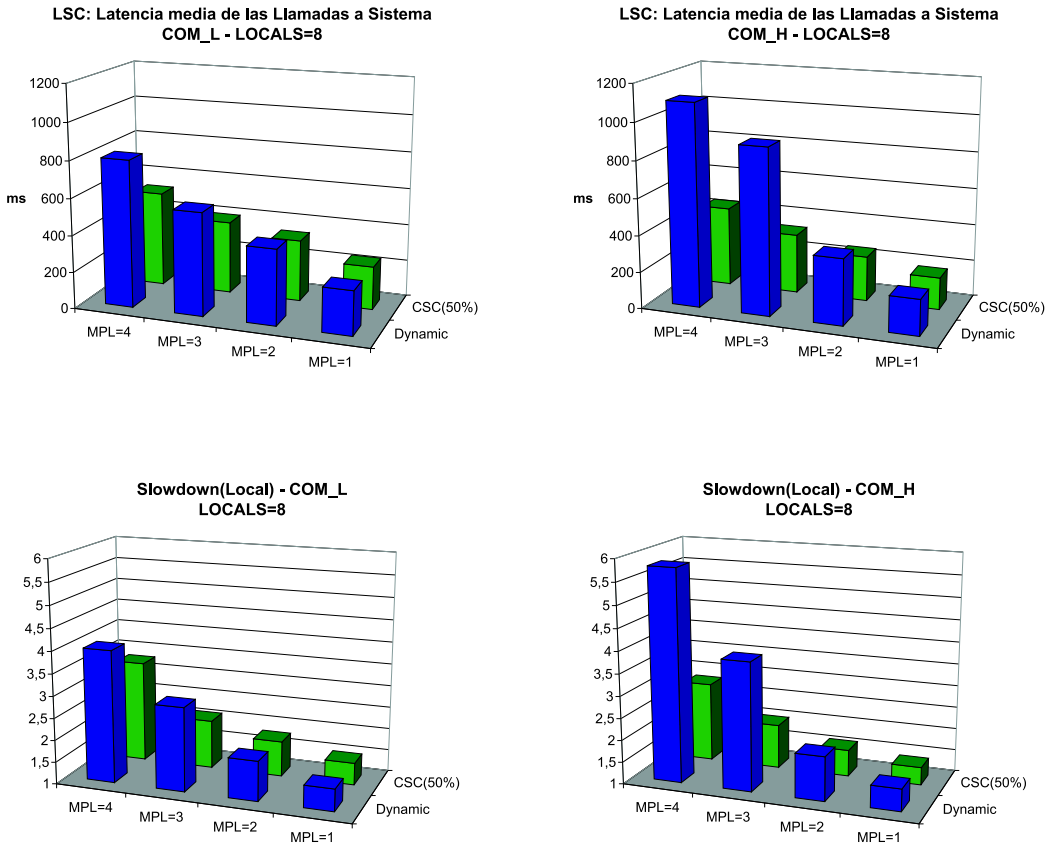

Figura 5.10: Latencia de las llamadas a sistema (arriba) y slowdown de los usuarios locales (abajo).

de CSC conlleva, para el caso de la carga con altos requerimientos de cómputo (COM_L), un incremento, con respecto a los valores obtenidos en el cluster homogéneo, de la latencia media de las llamadas a sistema, así como del slowdown asociado con los trabajos del usuario local, tal como ilustra la figura 5.10. Sin embargo, tanto el slowdown como la latencia se mantienen, para todos los casos estudiados con un $MPL < 4$, por debajo de los umbrales mínimos aceptados por el usuario local. Asimismo, el overhead introducido por CSC continua siendo ostensiblemente inferior al causado por Dynamic. Con respecto a los resultados obtenidos con la carga COM_H, esta diferencia entre CSC y Dynamic aumenta considerablemente debido a que CSC asigna la porción de memoria, asociada a las tareas paralelas, con respecto al menor tamaño de memoria principal presente a lo largo del cluster ($128MB$), hecho que comporta que dicha porción de memoria, para un porcentaje $L = 50\ \%$,

sea de $64MB$. De esta manera, los usuario locales trabajando en aquellos nodos con mayor tamaño de memoria dispondrán de una porción de memoria superior a la que en un principio deberían tener; hecho que repercute favorablemente en aquellas métricas asociadas con el usuario local y para aquellas cargas distribuidas con altos requerimientos de memoria.

5.3. Evaluación de CSC en un Entorno NOW de Producción

Con objeto de poder evaluar el comportamiento de CSC en un entorno cluster con usuarios locales reales, varias cargas distribuidas fueron ejecutadas en el laboratorio de prácticas *L1* de la *Escuela Politécnica Superior* (EPS) de la *Universitat de Lleida* (UdL). Este laboratorio se caracteriza por el hecho de que es utilizado tanto como laboratorio de prácticas cerradas, bajo la supervisión de un profesor, así como laboratorio abierto a los usuarios para que estos puedan realizar libremente sus prácticas.

El propósito de estas pruebas es evaluar el rendimiento de CSC desde una doble óptica:

1. Desde el punto de vista del usuario local se pretende medir la correlación entre las métricas obtenidas a partir del modelo de carga local (latencia media de llamadas a sistema y slowdown) empleado en el entorno controlado con respecto a la percepción real de los usuarios locales; percepción que dependerá tanto del grado de multiprogramación paralela, como del tipo de carga distribuida y del tipo de trabajo realizado por el usuario local.

2. En relación a las cargas distribuidas, estas pruebas permitirán medir el rendimiento de las mismas con respecto a diferentes escenarios, representativos de las diferentes actividades que se realizan cotidianamente en un laboratorio típico de prácticas de informática. Cabe decir que la doble funcionalidad del laboratorio empleado en esta experimentación difiere del modelo de carga local simulado en el escenario controlado. En las pruebas realizadas en el entorno controlado se había modelado el comportamiento de los usuarios de un laboratorio abierto al público, en el cual cada uno de los usuarios del mismo podía realizar un trabajo totalmente diferente. En cambio, estas nuevas pruebas serán realizadas tanto en entornos abiertos al público, como en entornos cerrados donde todos los usuarios hacen aproximadamente lo mismo, de acuerdo con las pautas marcadas por un guión de prácticas.

Atendiendo a dichos propósitos, en este entorno se ejecutaron las dos cargas extremas, definidas en la tabla 5.3, una con altos requerimientos de cómputo (COM_L), y la otra con bajos requerimientos de cómputo y altos de comunicación (COM_H). Asimismo, con objeto de no influir en la percepción del usuario local, estos no fueron informados respecto al doble uso que se realizaba de sus respectivos puestos de trabajo.

En estas pruebas se evaluó el rendimiento de CSC con respecto al coscheduling Dinámico tradicional. La política del coscheduling Dinámico fue tomada como referencia atendiendo a los resultados obtenidos con respecto al resto de técnicas de coscheduling tradicionales evaluadas, tanto en el entorno controlado como en el entorno simulado. CSC fue sintonizado, atendiendo a la experimentación previa realizada en el cluster controlado, con los siguientes parámetros: L=50 %, MEM_MIN=95 % y STEP=5.

Las pruebas se realizaron sobre 16 PCs del laboratorio *L1* de la *EPS (UdL)*. Cada uno de los puestos de trabajo de esta sala dispone de un procesador *PentiumIV* a 1800Mhz, con 256MB de memoria RAM, 256MB de Swap y con un disco de 20GB. Cada PC tiene instalada la versión Red-Hat 7.2 (kernel v.2.2.15) junto con el entorno PVM-3.4. Los diferentes PCs están interconectados mediante una red Fast Ethernet a 100Mbps.

La tabla 5.8 muestra los tiempos de ejecución de cada uno de los benchmarks que forman las cargas distribuidas mencionadas cuando fueron ejecutados tanto secuencialmente en una única máquina monoprocesador, como en paralelo, sin ningún usuario local interfiriendo la ejecución, y con un tamaño igual a 16 tareas. Este tamaño de 16 tareas fue escogido atendiendo a las siguientes razones:

1. El número de puestos de trabajo que cuenta dicho laboratorio determina el tamaño máximo posible. Teniendo en cuenta que el laboratorio *L1* dispone de 25 puestos de trabajo y que la gran mayoría de benchmarks de la NAS trabajan con un número de tareas que ha de ser potencia de dos, el número 16 es la potencia de dos más próxima al tamaño máximo del laboratorio.

2. En principio, atendiendo a los estudios realizados por *Leutenegger* et al. [LS97], cuánto menor es el tamaño de una aplicación menos posibilidades tiene que sea retardada debido a la presencia de un usuario local. Sin embargo, de acuerdo con los resultados obtenidos en la experimentación realizada en el cluster controlado con 4, 8 y 16 nodos, en general se cumple que cuánto menor es el tamaño de una aplicación mayores son los recursos de CPU y memoria que consume, con lo que se limita el grado de multiprogramación. Asimismo, estos mismos resultados muestran que la penalización provocada por la presencia de

un usuario local es menor que el beneficio provocado por incrementar el grado de multiprogramación. Teniendo en cuenta ambos factores se llega a la conclusión de que el número óptimo de tareas, para este caso concreto, es 16.

A diferencia del entorno controlado, donde cada carga estaba formada por 24 aplicaciones, que se corresponde con un tiempo máximo de ejecución de la carga alrededor de 45 minutos en este caso fue ampliado el número de benchmarks que integran la carga hasta 48, de modo que la duración máxima de su ejecución estuviese alrededor de las 2 horas, que se corresponde con el tiempo asignado a una sesión de prácticas de laboratorio. De este modo, los resultados obtenidos no se ven excesivamente afectados por los picos de cargas de trabajo puntuales de los usuarios locales.

Benchmarks	**Tiempos de Ejecución (s)**	
	1	**16**
IS.B	741	104
FT.A	454	128
CG.B	1289	258
EP.A	250	25
EP.B	1197	105
LU.A	348	39
BT.B	551	115
SP.A	448	75
MG.B	1280	104
Sintree-L	920	94
Sintree-H	620	84

Tabla 5.8: Tabla de resultados de los benchmarks distribuidos en el cluster de producción dedicado, cuando fueron ejecutados tanto secuencialmente en una única máquina monoprocesador, como en el cluster dedicado y con un tamaño igual a 16 tareas..

5.3.1. Escenarios de experimentación

El laboratorio *L1* [(EP04], empleado en esta experimentación, está abierto desde las 8 de la mañana hasta las 9 de la noche, durante los cinco días laborables de la semana. Estas 65 horas semanales se distribuyen de la siguiente manera: 28 horas (43 % del tiempo) abierto al publico y 37 horas (57 %) ocupado para la realización de prácticas dirigidas.

Atendiendo a los horarios de dicho laboratorio, las distintas prácticas que se realizan pueden ser agrupadas, de acuerdo con los recursos de cómputo utilizados por los usuarios locales, en tres diferentes escenarios. La tabla 5.9 muestra los recursos utilizados, por parte de los usuarios locales, en cada uno de los escenarios definidos. La columna indicada como *%máquinas_usadas* se corresponde con el porcentaje de máquinas ocupadas por los usuarios locales, mientras que las columnas *Carga_CPU* y *%Memoria* indican la carga media de la cola de preparados en el último minuto y el porcentaje medio de memoria ocupada, respectivamente. Entre paréntesis se muestra la desviación estándar calculada.

El primer entorno, el cual será denominado *Lab_Open*, se corresponde con aquellas horas donde el laboratorio está abierto para que los estudiantes puedan realizar las prácticas. Las pruebas realizadas en este escenario fueron realizadas durante las tres primeras semanas del mes de febrero en periodos de dos horas consecutivas. En la tabla 5.9 se observa como el grado de utilización de recursos es muy similar al perfil *Xwin* de usuario local (ver apéndice E) utilizado para simular a los usuarios locales en el entorno controlado. El interés de este escenario radica, en gran medida, en la alta variabilidad de los recursos utilizados por los usuarios del mismo.

Escenarios	**%máquinas_usadas**	**Carga_CPU**	**%Memoria**
Lab_Open	45 (20)	0.15(0.25)	40(50)
Lab_Estadis	100(0)	0.30(0.12)	60(10)
Lab_EDI	100(0)	0.10(0.05)	35(7)

Tabla 5.9: Grado de utilización medio del laboratorio *L1* durante los intervalos en que se realizaron las pruebas. Entre paréntesis se muestra la desviación obtenida.

El segundo escenario, denominado *Lab_Estadis*, se corresponde con las prácticas tutorizadas correspondientes a la asignatura de *Estadística* de tercer curso de la titulación de *Ingeniería técnica en Informática de Gestión*. Estas pruebas fueron realizadas dentro del horario de laboratorio asociado con dicha asignatura durante de las cuatro primeras semanas del segundo semestre del curso 2003/04. En estas prácticas, los estudiantes utilizan un software para cálculos estadísticos, denominado *R* [Uni02], el cual es ejecutado sobre el entorno de ventanas de Linux. La realización de las prácticas suponía el tratamiento estadístico de ficheros de datos, junto con la generación de diferentes gráficos. Este hecho comporta, como se refleja en la tabla 5.9, un uso relativamente importante de la CPU y de la memoria. Estos dos factores lo convierten, a priori, en un escenario poco propicio

para la ejecución de aplicaciones paralelas; por lo cual puede ser considerado como un escenario extremo para el análisis de los resultados obtenidos. Cabe decir que este escenario es representativo de todas aquellas prácticas que requieren un importante uso de la CPU, como por ejemplo podríamos citar las prácticas realizadas con calculadores simbólicos, como el *Mathematica* [Wol03], utilizado en varias asignaturas de cálculo numérico o bien programas de procesamiento numérico, como el *MatLab* [Mat03], utilizado en asignaturas de procesamiento de señal.

El tercer escenario, *Lab_EDI*, se corresponde con las prácticas tutorizadas de los estudiantes de programación de segundo curso de la titulación de I*ngeniería técnica en Informática de Gestión*. Estas pruebas fueron realizadas dentro del horario asociado con la asignatura de *Estructura de Datos y de la Información* (EDI) durante de las cuatro primeras semanas del segundo semestre del curso 2003/04. Mediante un sencillo editor, normalmente *emacs* o bien *kwrite*, los estudiantes editan programas en C++ y posteriormente los compilan mediante el compilador estándar de Linux *gcc*. La poca embergadura de los programas realizados hacen que la carga de CPU y los porcentajes de uso de memoria sean relativamente pequeños. Aunque los recursos utilizados por los usuarios de este escenarios son similares a los del entorno *Lab_Open* previamente definido, la escasa variabilidad de los mismos convierten a *Lab_EDI* en un escenario propicio para la ejecución de las aplicaciones distribuidas y, por tanto, puede ser considerado como un caso extremo positivo para los intereses del usuario de aplicaciones distribuidas. Finalmente, cabe destacar que este último escenario es representativo de una gran variedad de asignaturas asociadas con la rama de programación.

5.3.2. Resultados experimentales obtenidos

La metodología seguida en esta experimentación ha consistido en monitorizar la *carga de CPU* (número medio de tareas en la cola de preparados durante el último minuto) y el *porcentaje de memoria demandada* por todos los procesos, tanto locales como distribuidos, que se ejecutan en cada uno de los nodos del cluster. De este modo, el grado de multiprogramación de las aplicaciones distribuidas ha sido incrementado gradualmente hasta alcanzar los valores máximos de carga de CPU y porcentaje de memoria permitidos por el usuario local. De acuerdo con el estudio descrito en la sección B.4 del apéndice B, donde se muestra la correspondencia entre las dos métricas utilizadas para medir el grado de satisfacción de los usuario locales, slowdown y latencia de llamadas a sistema, con respecto a la carga de CPU y al porcentaje de memoria monitorizados, los valores máximos de estos dos parámetros de sistema monitorizados han sido fijados en 1,8 y 85 % para la

política Dinámica y 2,2 y 125 % para la política CSC, respectivamente.

Asimismo, con el objetivo de verificar que las aplicaciones de los usuarios locales no se lentifican por debajo de los mínimos aceptables, diferentes aplicaciones locales han sido monitorizadas tanto en aquellos nodos compartidos con las aplicaciones distribuidas, como en aquellos nodos dedicados al usuario local. Acorde con estos propósitos se ha monitorizado el proceso de compilación CPU intensivo *(gcc),* utilizado por los usuarios del laboratorio *Lab_EDI,* así como el proceso estadístico interactivo *(R),* utilizado por los usuarios del laboratorio *Lab_Estadis.*

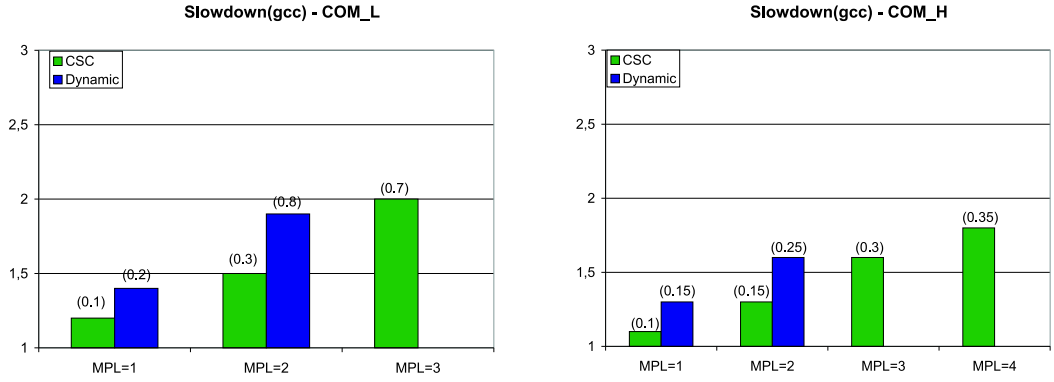

Figura 5.11: Slowdown del proceso *gcc* con la carga *COM_L* (izquierda) y *COM_H* (derecha) para los diferentes grados de multiprogramación y políticas evaluadas. Entre paréntesis se muestra la desviación típica asociada.

Con respecto al proceso *gcc* se ha calculado el *slowdown* de acuerdo con la siguiente expresión:

$$Slowdown(gcc) = \frac{Tejec(gcc, node^{no\_dedicado})}{Tejec(gcc, node^{dedicado})}, \tag{5.5}$$

donde $Tejec(gcc, node^{no\_dedicado})$ y $Tejec(gcc, node^{dedicado})$ son los tiempos de ejecución del proceso *gcc* en un nodo no dedicado y dedicado, respectivamente. La figura 5.11 muestra el slowdown medio asociado al proceso *gcc* cuando éste fue ejecutado junto con la carga *COM_L* y *COM_H*, respectivamente. Asimismo, entre paréntesis se muestra la desviación estándar obtenida. Estos resultados muestran como los valores máximos de carga de CPU y memoria permitidos por el usuario local son alcanzados para un $MPL = 2$ para el caso de la política Dynamic, mientras que la política CSC permite incrementar el grado de multiprogramación hasta 3 para la carga *COM_L* y 4 para la

carga COM_H, respectivamente. En general se observa como la evolución, para ambas cargas, es muy similar a la que se obtuvo en el entorno controlado. En todos los casos evaluados, el slowdown es inferior al umbral fijado de dos, resultado que reafirma la correlación existente entre las métricas de usuario local empleadas con respecto a los parámetros de sistema monitorizados (ver sección B.4 del apéndice B). Los resultados obtenidos muestran como el slowdown introducido por CSC, para un mismo grado de multiprogramación, es aproximadamente la mitad que el introducido por Dynamic; hecho que permite a CSC incrementar el grado de multiprogramación paralela y, por tanto, la eficiencia de uso del sistema. Asimismo, el análisis de las desviaciones obtenidas reflejan la estabilidad alcanzada por la política CSC debido a la limitación de los recursos de cómputo asignados a las aplicaciones distribuidas. Cabe resaltar que en el caso de la carga COM_L, el grado de multiprogramación permitido por el usuario local es mayor que el máximo alcanzado en el entorno controlado, tanto para el caso de CSC, donde se consigue un $MPL = 3$, como para el Dynamic con un $MPL = 2$. Este incremento es debido a que las máquinas de este laboratorio tienen unas prestaciones superiores que las del cluster controlado, hecho que repercute en una menor carga de CPU media, y por tanto en un incremento de los recursos disponibles para el usuario de las aplicaciones distribuidas. Estos resultados reafirman la necesidad de utilizar políticas de planificación que tengan en cuenta la actividad del usuario local, como es el caso de CSC, y que limiten el overhead causado en el trabajo del usuario local.

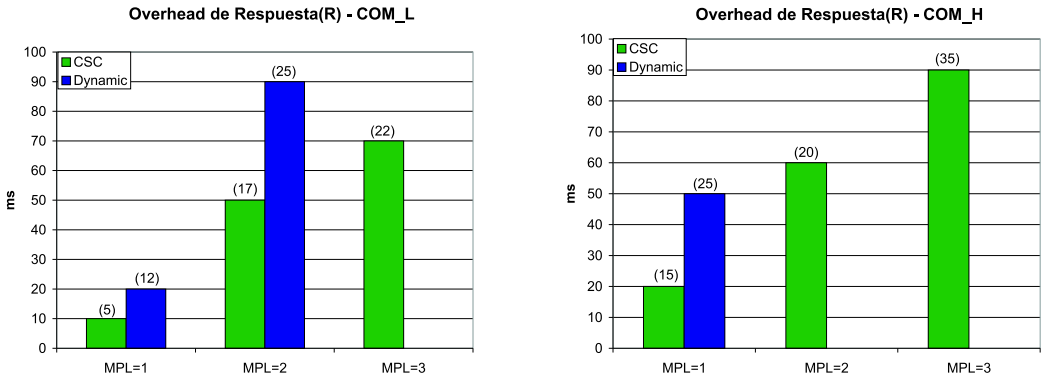

Figura 5.12: Overhead de Respuesta de la aplicación interactiva R cuando fue ejecutada junto con la carga COM_L (izquierda) y COM_H (derecha). Entre paréntesis se muestra la desviación típica asociada.

Asimismo, con objeto de medir el tiempo de respuesta de los procesos

interactivos, se ha calculado el *overhead de respuesta* para la aplicación estadística R. El *overhead de respuesta* es definido como el tiempo medio que consume el proceso monitorizado en ser planificado desde la última vez que ha sido insertado en la cola de preparados. Los resultados mostrados en la figura 5.12 muestran como, en todos los casos estudiados, el tiempo de respuesta obtenido es menor que el umbral máximo fijado de $400ms$. De hecho, todos los casos analizados con la carga COM_L dan un tiempo de respuesta menor que $100ms$. Este bajo overhead es debido a que cada vez que el proceso interactivo es insertado en la cola de preparados tiene una prioridad de planificación superior que el resto de procesos distribuidos residentes en el nodo, hecho que comporta que el proceso local R pueda apropiarse de la CPU. Por otro lado, el tiempo de respuesta obtenido en todos los casos estudiados con la carga COM_H es mayor que el asociado a la carga COM_L. Esta diferencia de comportamiento es debida a que, en el caso de la carga COM_H, la prioridad de planificación de las tareas distribuidas que se comunican es incrementada, tanto con la política CSC como con la Dynamic, con el objetivo de alcanzar el coscheduling de las tareas que se comunican entre si, hecho que comporta que, en algunas ocasiones, las aplicaciones locales no tengan una prioridad suficientemente grande como para poder apropiarse de la CPU. Asimismo, al igual que ocurría con la métrica slowdown, CSC obtiene unos resultados mucho mejores que Dynamic, tanto en lo que respecta a las cotas de intrusión como al grado de multiprogramación alcanzado. En este sentido cabe destacar que bajo la política Dynamic, la carga COM_H solamente ha podido ser ejecutada con un $MPL = 1$ debido a los elevados requerimientos de memoria de la carga distribuida y a la alta variabilidad en la demanda de los recursos de memoria asociada a los usuarios locales de este determinado entorno. Comparando estos resultados con los obtenidos para la métrica *latencia de las llamadas a sistema* utilizada en el entorno controlado, para medir el tiempo de respuesta de un usuario interactivo, se observa como la evolución de ambas métricas es muy similar, de modo que el grado de multiprogramación máximo alcanzado es exactamente el mismo que en el entorno controlado.

La figura 5.13 muestra el máximo speedup obtenido al ejecutar las cargas COM_L (figura 5.13(izq.)) y COM_H (figura 5.13(der.)) en cada uno de los tres escenarios anteriormente definidos. Asimismo, se ha añadido, a modo de referencia, el speedup obtenido al ejecutar ambas cargas en el cluster dedicado, es decir, sin ningún usuario local interfiriendo la ejecución de las aplicaciones paralelas. En ambos gráficos se observa una importante dependencia de los resultados obtenidos con respecto al tipo de laboratorio utilizado. El análisis de los resultados obtenidos en los tres escenarios definidos en el cluster no dedicado muestra como los mejores resultados se obtienen

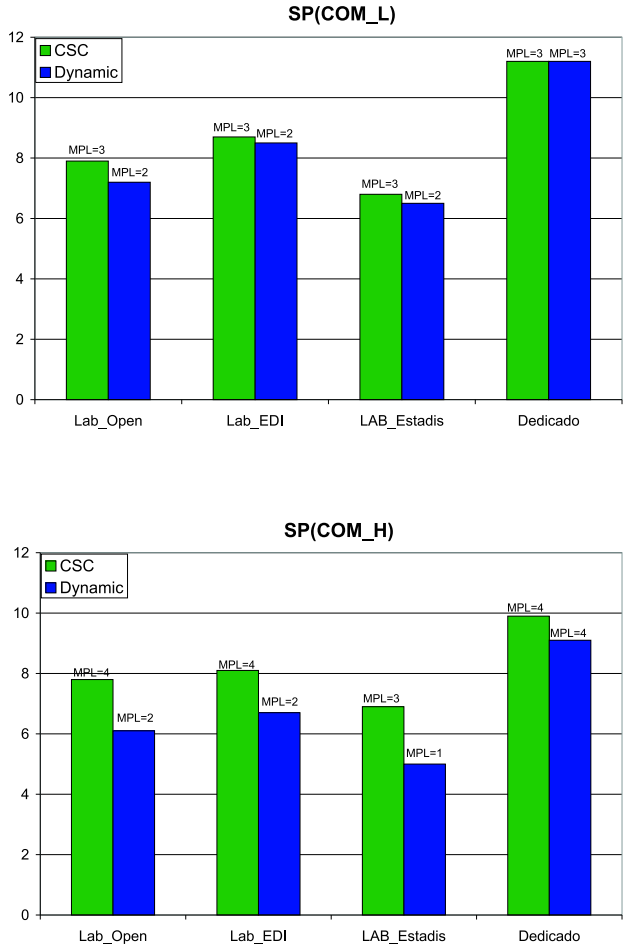

Figura 5.13: Speedup de las cargas distribuidas en un entorno cluster productivo. En el *top* de cada barra se muestra el MPL asociado al speedup obtenido.

para ambas cargas en el entorno *Lab_EDI*, básicamente debido a la baja utilización de los recursos por parte de los usuarios de este tipo de laboratorio, mientras que los peores resultados son obtenidos, como cabía esperar, en el laboratorio *Lab_Estadis*. En el caso de la carga *COM_L* se observa como en los entornos con un gran número de usuarios locales, como es el caso de *Lab_EDI* y *Lab_Estadis*, ambas políticas evaluadas obtienen un resultado similar, con una ligera ganancia para CSC. Esta ganancia es debida a que CSC rebaja el quantum de las tareas distribuidas a la mitad, en todos aquellos nodos donde hay actividad de usuario local; hecho que le permite

incrementar el grado de multiprogramación hasta tres y, en consecuencia, obtener un speedup final ligeramente mejor. Con respecto a los resultados obtenidos con la carga *COM_H*, cabe destacar como Dynamic, debido a los altos requerimientos de memoria de esta carga distribuida, solamente es capaz de alcanzar un máximo $MPL = 2$, en el mejor de los casos, mientras que CSC consigue alcanzar un $MPL = 4$, tanto en el laboratorio *Lab_Open* como en el laboratorio *Lab_EDI*. Este aumento de MPL se traduce en una mejora considerable con respecto al speedup obtenido. Finalmente, comparando los resultados alcanzados en los tres escenarios asociados al cluster no dedicado con respecto a los valores obtenidos en el cluster dedicado, se observa como la máxima diferencia obtenida (carga *COM_L* y escenario *Lab_Estadis*) es de aproximadamente 4,5 puntos. En este sentido es importante remarcar que el speedup obtenido en la ejecución sobre el cluster no dedicado, supone un coste cero a la institución o empresa propietaria del cluster, dado que se están utilizando unos recursos ya disponibles.

En resumen, la experimentación realizada en el entorno productivo ha permitido reafirmar las tendencias estudiadas en el entorno controlado, de modo que CSC obtiene unos resultados globales mejores que el resto de políticas evaluadas, tanto en lo que respecta al rendimiento de las aplicaciones del usuario local como de las aplicaciones distribuidas. Esta ganancia de la política CSC es debida a su capacidad de adaptar dinámicamente los recursos asignados a la carga distribuida en función de la variación de los recursos requeridos por los usuarios locales, hecho que permite a CSC trabajar con unos grados de multiprogramación paralelos superiores, alcanzándose en algunos casos una relación de 4 a 2, y, aumentar, de este modo, la eficiencia con respecto al grado de utilización de los recursos de cómputo del cluster. Asimismo, estos resultados demuestran que las políticas de prevención del rendimiento del usuario local, utilizadas por CSC, no son en absoluto antagónicas con respecto al rendimiento de las tareas distribuidas, de modo que ambos objetivos pueden complementarse, siempre y cuando las decisiones de planificación sean tomadas atendiendo tanto a las necesidades, a nivel de proceso, de cada uno de los nodos del cluster, como a las necesidades a nivel de aplicación, tomadas de acuerdo con una política global a lo largo del cluster. Finalmente, cabe destacar que los resultados obtenidos por CSC demuestran que el desarrollo de políticas de planificación a corto plazo orientadas a cluster no dedicados permiten explotar, de una manera eficiente, todos aquellos recursos de cómputo disponibles, obteniendo unas métricas de speedup satisfactorias y provocando un overhead inapreciable para el usuario local. Cabe resaltar que dicho beneficio es obtenido sin coste económico adicional alguno, es decir, aprovechando unos recursos ya disponibles en la empresa o institución correspondiente.

Capítulo 6

Conclusiones y Principales Contribuciones

El objetivo de este trabajo de tesis ha sido contribuir en la resolución del problema del coscheduling de aplicaciones distribuidas en un entorno cluster no dedicado; donde cada nodo del cluster puede ser compartido entre las aplicaciones de un usuario local y las aplicaciones distribuidas. Estos entornos, debido principalmente a la amplia gama de trabajos que puede llegar a realizar un usuario local, se caracterizan por una alta variabilidad de los recursos de cómputo disponibles, hecho que provoca que las tradicionales técnicas de coscheduling, basadas exclusivamente en el análisis de los eventos locales de comunicación, no puedan garantizar un rendimiento eficiente tanto de las aplicaciones locales como de las aplicaciones distribuidas.

Este nuevo marco de trabajo ha comportado un replanteamiento del problema del coscheduling de aplicaciones distribuidas. En nuestro modo de ver, el fin del coscheduling no solamente debe estar restringido a decidir *cuándo* deben ser asignados los recursos de cómputo a las aplicaciones distribuidas, caso de las técnicas de coscheduling tradicionales, si no también *cuántos* recursos son asignados para cada aplicación. Este doble propósito nos ha llevado a desarrollar en este trabajo de tesis una nueva propuesta de coscheduling, denominada *CoScheduling Cooperativo* (*CS*C), orientada a la coordinación de múltiples aplicaciones paralelas en un entorno no dedicado.

CSC, a diferencia de las propuestas de coscheduling tradicionales, gestiona los recursos de cómputo de cada nodo, tanto en función de la ocurrencia de determinados eventos locales (de memoria, de CPU, de comunicación y de actividad del usuario local), como de la recepción de aquellos eventos ocurridos en nodos remotos y que han provocado una modificación de los recursos asociados a aquellos procesos distribuidos que cooperan entre si (*procesos cooperantes*). El análisis de estos eventos permite a CSC adaptar los recur-

sos de cómputo del cluster a las necesidades de ambos tipos de usuarios; el usuario local, caracterizado por unos elevados requerimientos de interactividad, y el usuario paralelo, en el cual priman los requerimientos de cómputo y de comunicación. A nivel local, CSC prioriza la asignación tanto de la CPU, como de la memoria, al usuario local, garantizando unos tiempos de respuesta de sus aplicaciones dentro de unos márgenes aceptables para el mismo. El resto de recursos locales de cada nodo son asignados por CSC atendiendo tanto a las necesidades de CPU, de memoria y de comunicación de las tareas distribuidas residentes en dicho nodo, como a los requerimientos de las tareas cooperantes ejecutándose en nodos remotos. De este modo, CSC consigue balancear los recursos asignados a las tareas que forman una misma aplicación distribuida a lo largo del cluster y, por consiguiente, mejorar la coordinación entre las tareas que integran una misma aplicación.

Esta nueva propuesta de coscheduling ha sido realizada de acuerdo con los siguientes puntos relacionados con el desarrollo del modelo de coscheduling, implementación y validación de la misma. En cada uno de los puntos se citan además las aportaciones a que han dado lugar.

Respecto al modelo de coscheduling, se han analizado los modelos clásicos de coscheduling, junto con las políticas desarrolladas a partir de los mismos. El rendimiento de cada política ha sido evaluado, tanto mediante simulación como por medio de experimentación, en un entorno real controlado, en función de diferentes parámetros que caracterizan un entorno cluster no dedicado. Los resultados obtenidos en dicha experimentación han puesto de manifiesto que una política de coscheduling, orientada a esta clase de entornos, ha de tener en cuenta, además de los eventos de comunicación contemplados por las políticas tradicionales de coscheduling, diferentes aspectos adicionales como son los requerimientos de memoria y de CPU, tanto de las aplicaciones locales como de las distribuidas, la longitud del quantum, la actividad de los usuarios locales, el grado de multiprogramación de las aplicaciones paralelas, así como el balanceo de recursos de cómputo a lo largo del cluster.
Atendiendo a estos resultados, un nuevo modelo de coscheduling que contempla los parámetros anteriores ha sido definido. La definición formal de este nuevo modelo de coscheduling, junto con un análisis por medio de la simulación de las ventajas que reporta, ha sido publicado en:

[GSHL01a] F. Giné, F. Solsona, P. Hernández y E. Luque. *Coscheduling under Memory Constraints in a NOW Environment.* In 7th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP'01), volume 2221 of Lecture Notes in Computer Science, pages 41-65, 2001.

A partir de este nuevo modelo, diferentes modificaciones de las políticas de coscheduling tradicionales, fruto tanto de diferentes referencias de la literatura como de nuestras propias intuiciones han sido analizadas por medio de simulación. El análisis de los resultados obtenidos ha permitido el desarrollo de esta nueva propuesta de coscheduling, denominada *Coscheduling Cooperativo (CSC).*

Respecto a la política CSC, se han definido, a nivel algorítmico, e implementado en un entorno Linux-PVM, cada uno de los módulos que la integran. Estos módulos se encargan de la captura y análisis tanto de los eventos locales de comunicación (emisión y recepción de mensajes), de memoria (fallos de página), de actividad del usuario local (teclado y ratón), CPU (estado de la cola de preparados y prioridad de planificación), así como de eventos de actividad de usuario y memoria ocurridos en nodos remotos. A partir del análisis de dichos eventos, estos módulos gestionan la *prioridad de planificación*, la *longitud del quantum*, la *asignación de memoria* a las tareas, tanto locales como distribuidas, así como el *balanceo de recursos* de CPU y memoria asignados a las aplicaciones distribuidas que se ejecutan a lo largo del cluster.

El *módulo de gestión de memoria* se encarga de asegurar, tanto a las tareas locales como distribuidas, un porcentaje de memoria suficiente para que éstas puedan progresar eficientemente, de modo que los usuarios, tanto locales como distribuidos, conozcan a priori los recursos de memoria disponible y, en función de los mismos, puedan ejecutar sus correspondientes cargas.

En la misma línea mostrada por diferentes trabajos previos al desarrollo de esta tesis, los resultados experimentales obtenidos en esta tesis revelan que el rendimiento de las tareas distribuidas puede caer drásticamente ($Slowdown > 10$) debido al efecto de una excesiva tasa de fallos de página provocada por un desbordamiento de la memoria principal en uno o varios nodos del cluster; efecto que se agrava a medida que aumentan los requerimientos de comunicación de las tareas distribuidas. Con el objetivo de minimizar esta excesiva paginación y, por consiguiente mejorar la coordinación entre las tareas distribuidas que se comunican entre si, el *módulo de gestión de memoria* interacciona con el *módulo de asignación de prioridades de planificación.* De este modo, la prioridad de las tareas distribuidas es asignada atendiendo tanto a la tasa de fallos de página, como a la ratio de comunicación de las tareas distribuidas. En aquellos nodos donde la memoria es desbordada, CSC prioriza la ejecución de las tareas distribuidas con menor tasa de

fallos de páginas, parando y desalojando de memoria a la aplicación distribuida con un mayor requerimiento de memoria; mientras que en los nodos donde la memoria no es desbordada, CSC asigna las prioridades de las tareas distribuidas atendiendo a la tasa de comunicación de las mismas.
Los algoritmos del *módulo de gestión de memoria* y de *asignación de prioridades*, junto con sus respectivas implementaciones en el s.o. Linux, han sido publicados en el siguiente articulo. En esta misma publicación se ha mostrado experimentalmente el buen rendimiento de la propuesta realizada, con respecto a las políticas de coscheduling más significativas mostradas en la literatura, tanto en lo que respecta a la percepción del usuario local, como al punto de vista del usuario de las aplicaciones distribuidas.

[GSHL02c] F. Giné, F. Solsona, P. Hernández y E. Luque. *Minimizing Paging Tradeoffs Applying Coscheduling Techniques in a Linux Cluster.* In 5th Vecpar and Parallel Processing Conference, Lecture Notes in Computer Science, volume 2565 of Lecture Notes in Computer Science, pages 592-607, 2002.

Diferentes trabajos previos a la realización de esta tesis han puesto de manifiesto la relación existente entre la longitud del quantum y el rendimiento de las políticas de coscheduling aplicadas en entornos cluster dedicados. Atendiendo a estos resultados, la política CSC ha integrado un algoritmo de asignación dinámica de la longitud del quantum. Al igual que en la política de asignación de la prioridad de planificación, el *módulo de asignación de quantum* aplica una política diferente atendiendo al estado de la memoria. De este modo, en caso de que la memoria no sea desbordada, la longitud del quantum es asignada teniendo en cuenta tanto la presencia del usuario local, con el objetivo de fijar el máximo porcentaje de cómputo asignado a las tareas distribuidas y de este modo limitar el retardo introducido en el trabajo del usuario local, como el estado global de todas las tareas que forman un mismo trabajo distribuido. Asimismo, esta asignación del quantum se realiza atendiendo al grado de heterogeneidad del cluster, extendiendo de este modo, el uso de la política CSC tanto a entornos homogéneos como heterogéneos. Por otro lado, en el caso de que la memoria de un determinado nodo sea desbordada y no haya actividad de usuario local, la longitud del quantum original es incrementada hasta un valor máximo, fijado por el administrador del sistema, con el objetivo de explotar la

localidad de las tareas distribuidas y de este modo disminuir el número de fallos de página.
En la siguiente publicación se ha mostrado un estudio respecto el grado de influencia de la longitud del quantum, tanto en lo que respecta al rendimiento de las aplicaciones locales, como de las distribuidas. Este estudio incluye un análisis respecto el grado de influencia de la memoria cache con respecto al rendimiento de las aplicaciones distribuidas. Los resultados obtenidos han mostrado que en aquellos nodos en los cuales no existen usuarios locales, el rendimiento de la cache aumenta para valores de quantum alrededor de $800ms$, hecho que repercute en una ligera ganancia en los tiempos de ejecución de las correspondientes aplicaciones distribuidas. Asimismo, estos resultados han revelado que esta ganancia es muy susceptible del patrón y frecuencia de comunicación de las correspondientes aplicaciones distribuidas, de modo que disminuye a medida que aumentan las frecuencias de comunicación de las tareas distribuidas. Además, esta ganancia desaparece en aquellos entornos en los cuales existe algún usuario local. Atendiendo a los resultados obtenidos, en este articulo se ha presentado el algoritmo de asignación de la longitud del quantum aplicado por la política CSC.

[GSHL02b] F. Giné, F. Solsona, P. Hernández y E. Luque. *Adjusting Time Slices to Apply Coscheduling Techniques in a Non-dedicated NOW.* In Euro-Par'2002 Conference, volume 2400 of Lecture Notes in Computer Science, pages 234-240, 2002.

Una extensión del algoritmo de asignación de quantum aplicado en entornos con altos requerimientos de memoria ha sido mostrado en la siguiente publicación. Los resultados obtenidos en este articulo han mostrado como una longitud del quantum alrededor de $1000ms$, puede llegar a reducir el índice medio de fallos de página a la mitad; hecho que comporta una disminución importante del slowdown de las tareas distribuidas (30 %).

[GSHL02a] F. Giné, F. Solsona, P. Hernández y E. Luque. *Adjusting the Length of Time Slices when Scheduling PVM Jobs with High Memory Requirements.* In EuroPVM/MPI'2002, volume 2474 of Lecture Notes in Computer Science, pages 156-164, 2002.

Con el objetivo de coordinar de un modo global las distintas decisiones

tomadas localmente en cada nodo, por los módulos descritos anteriormente, CSC ha incorporado el *módulo de gestión de recursos remotos*. Este módulo permite balancear los recursos de cómputo a lo largo del cluster, de manera que todas las tareas que pertenecen a una misma aplicación distribuida dispongan de una prioridad de asignación de recursos similar en cada uno de los nodos que integran el cluster y, de este modo, puedan progresar de un modo coordinado a lo largo del cluster. Con este fin, este bloque se encarga de enviar, recibir y gestionar todos aquellos eventos que han provocado cambios en la asignación de recursos en las tareas distribuidas residentes en nodos remotos, hecho que provoca una elevada interacción de este módulo con el resto de módulos que constituyen CSC. El funcionamiento de este módulo, junto con la interacción con el resto de módulos que integran CSC, así como su implementación en un entorno PVM-Linux ha sido descrita en la siguiente publicación.

[GSHL03a] F. Giné, F. Solsona, P. Hernández y E. Luque. *Cooperating Coscheduling in a Non-dedicated Cluster.* In Euro-Par'2003 Conference, volume 2790 of Lecture Notes in Computer Science, pages 212-218, 2003.

Respecto a la experimentación realizada, se ha comparado el rendimiento de CSC, con respecto a las políticas de coscheduling tradicionales más significativas, en dos diferentes tipos de NOW: un *entorno NOW controlado* y un *entorno NOW de producción*. El escenario controlado se caracteriza porque los requerimientos de la carga local han sido simulados mediante el uso de un benchmark sintético, el cual permite, al usuario del mismo, fijar tanto los requerimientos de comunicación, de memoria, de E/S a disco y de CPU. Estas pruebas han permitido sintonizar los diferentes parámetros de entrada de CSC en un amplio abanico de escenarios sintéticos, caracterizados por una alta variabilidad de recursos de cómputo utilizados tanto por los usuarios locales, como por los usuarios de las aplicaciones paralelas. De este modo, el comportamiento de CSC y del resto de políticas evaluadas ha sido cuantificado mediante un conjunto de métricas que permiten medir y comparar el rendimiento, tanto de las aplicaciones distribuidas como de las locales. La segunda parte de la experimentación ha sido realizada en un laboratorio real de usuarios (entorno NOW de producción), donde se han ejecutado las diferentes cargas distribuidas mientras los estudiantes estaban realizando sus correspondientes prácticas, ya sean cerradas o abiertas. Estas pruebas han permitido validar la bondad de

las métricas definidas en el entorno controlado, así como la correlación de las mismas con respecto a la percepción real tanto de los usuarios de las aplicaciones distribuidas como de las locales. Sobre ambos entornos se han ejecutado diferentes cargas distribuidas, caracterizadas por diferentes requerimientos, tanto de comunicación, de CPU, como de memoria.

El primer paso de la experimentación realizada en el entorno controlado ha consistido en evaluar el rendimiento, tanto de las tareas locales como de las distribuidas, en función del porcentaje de recursos asignados por CSC a las tareas distribuidas. Estos resultados han mostrado la necesidad de limitar los recursos utilizados por las tareas distribuidas, siendo un porcentaje del 50 % el que mejor se ajusta a las necesidades tanto del usuario local como del distribuido. En este sentido cabe destacar como CSC es capaz, aun disminuyendo los recursos asignados a las tareas distribuidas, de obtener un buen rendimiento para las mismas y, al mismo tiempo, conservar el rendimientos de las aplicaciones locales, dentro de unos límites aceptables por los usuarios locales. Asimismo, esta experimentación refleja como CSC obtiene los mejores resultados con aquellas cargas con un volumen de comunicación medio o alto, debido a su capacidad de coordinar los recursos asignados a las tareas distribuidas a lo largo de todo el cluster.

El siguiente paso de las experimentación realizada en el entorno controlado ha consistido en estudiar el rendimiento de las tareas locales y distribuidas en función del grado de multiprogramación de las aplicaciones paralelas. Los resultados obtenidos han revelado que aquellas políticas que no incorporan medidas de limitación de los recursos de cómputo utilizados por las tareas paralelas, como es el caso de las técnicas de coscheduling basadas en el modelo tradicional, el grado de multiprogramación de las mismas no puede ser superior a uno, en la mayoría de casos evaluados. En cambio, el comportamiento de CSC depende totalmente de las características de las aplicaciones distribuidas. Estos resultados han reflejado que para cargas con altos requerimientos de cómputo, el grado de multiprogramación no puede ser superior a dos, básicamente debido a que el slowdown del usuario local puede superar el umbral máximo fijado de dos, aunque el tiempo de respuesta del usuario local se mantenga muy por debajo de $400msg$, límite a partir del cual el usuario local percibe una ralentización de su trabajo. Sin embargo, para cargas con elevados requerimientos de comunicación, el grado de multiprogramación aplicado bajo CSC puede ser ampliado hasta un máximo de cuatro, verificando que CSC obtiene sus mejores resultados globales para cargas con altos requerimientos de comuni-

cación.
Finalmente, el rendimiento de CSC ha sido evaluado en un entorno cluster heterogéneo, tanto a nivel de tamaño de memoria como de potencia de procesador. Los resultados obtenidos han revelado como el rendimiento de las tareas locales y distribuidas mejora ostensiblemente con CSC debido a que esta política asigna el quantum y la memoria con respecto al grado de heterogeneidad de los nodos, balanceando de este modos los recursos de cómputo asignados a lo largo del cluster. Asimismo, estos resultados muestran como la ganancia de CSC, con respecto al resto de políticas evaluadas, se incrementa a medida que se aumenta tanto el grado de multiprogramación de las aplicaciones paralelas, como el número de usuarios locales presentes en el cluster.
Los resultados experimentales obtenidos en el cluster controlado, tanto en entornos homogéneos como heterogéneos, han sido publicados en los dos siguientes artículos. El primero de ellos ha sido enfocado al análisis del rendimiento de CSC en entornos con altos requerimientos de memoria, mientras que en el segundo se han mostrado aquellos resultados obtenidos con cargas que encajasen totalmente en el tamaño de la memoria principal.

[GSHL03b] F. Giné, F. Solsona, P. Hernández y E. Luque. *Dealing with Memory Constraints in a Non-dedicated Linux Cluster.* The International Journal of High Performance Computing Applications, volume 17, number 1, pages 39-48, 2003.

[GHS+03] F. Giné, M. Hanzich, F. Solsona, P. Hernández y E. Luque. *Multiprogramming Level of PVM Jobs in a Non-dedicated Linux NOW.* In EuroPVM/MPI'2003, volume 2840 of Lecture Notes in Computer Science, pages 577-586, 2003.

Con objeto de poder evaluar el comportamiento de CSC en un entorno cluster con usuarios locales reales, varias cargas distribuidas, con distintos requerimientos de comunicación y cómputo, han sido ejecutadas en un laboratorio de prácticas de la *Escuela Politécnica Superior* (EPS) de la *Universitat de Lleida* (UdL). Este laboratorio se caracteriza por el hecho de que es utilizado tanto como laboratorio de prácticas cerradas bajo la supervisión de un profesor, así como laboratorio abierto a los usuarios, para que estos puedan realizar libremente sus prácticas. Las cargas distribuidas han sido lanzadas durante intervalos de tiempo que fueran representativos de los diferentes usos que se hace de dicho

laboratorio.
Los resultados obtenidos en esta experimentación han reafirmado las tendencias mostradas en el entorno controlado, de modo que CSC obtiene unos resultados globales mejores que el resto de políticas evaluadas, tanto en lo que respecta al rendimiento de las aplicaciones del usuario local como al de las aplicaciones distribuidas. Esta ganancia de la política CSC es debida a su capacidad de adaptar dinámicamente los recursos asignados a la carga distribuida en función de la variación de los recursos requeridos por los usuarios locales, hecho que ha permitido a CSC trabajar con unos grados de multiprogramación paralelos superiores, alcanzándose en algunos casos una relación de 4 a 2, con respecto a las políticas de coscheduling tradicionales, conservando, al mismo tiempo, el rendimiento del usuario local. De este modo, la eficiencia de CSC con respecto al grado de utilización de los recursos de cómputo del cluster ha sido incrementada. Asimismo, estos resultados han demostrado que las políticas de preservación del rendimiento del usuario local, utilizadas por CSC, no son en absoluto antagónicas con respecto al rendimiento de las tareas distribuidas, de modo que ambos objetivos pueden complementarse, siempre y cuando las decisiones de planificación sean tomadas atendiendo tanto a las necesidades, a nivel de proceso, de cada uno de los nodos del cluster, como a las necesidades a nivel de aplicación, tomadas de acuerdo con una política global a lo largo del cluster. Finalmente, cabe destacar que los resultados obtenidos por CSC demuestran que el desarrollo de políticas de planificación a corto plazo orientadas a cluster no dedicados permiten explotar, de una manera eficiente, todos aquellos recursos de cómputo disponibles, obteniendo unas métricas de speedup favorables y provocando un overhead inapreciable para el usuario local. Cabe resaltar que dicho beneficio es obtenido sin coste económico adicional alguno, es decir aprovechando unos recursos ya disponibles en la empresa o institución correspondiente.

6.1. Líneas Abiertas

A partir de la experiencia adquirida en el desarrollo de esta tesis, se han ido viendo las siguientes nuevas líneas de actuación que pueden contribuir a completar el nivel de refinamiento y la amplitud del problema del coscheduling en entornos cluster no dedicados.

- Desarrollar una metodología para realizar predicciones de los recursos utilizados por los usuario locales, a partir de su historia pasada, y,

de este modo, poder estimar los recursos que dispondrán las tareas paralelas para su ejecución En esta misma línea, sería interesante realizar una estimación de la frecuencia de llegadas de las aplicaciones distribuidas, junto con los requerimientos de cómputo de las mismas. De este modo, la interacción entre ambas estimaciones puede ayudar a fijar el rendimiento de las aplicaciones distribuidas, dentro de unos determinados umbrales de calidad.

- Complementar la planificación a corto plazo realizada por CSC, con un planificador de trabajos a largo plazo que aplique una estrategia mixta de espacio y tiempo compartido, de modo que el mapping de las tareas distribuidas sea realizado atendiendo tanto al grado de utilización de los recursos del sistema por parte de los usuarios locales, como a los requerimientos de CPU, de memoria y de comunicación de las cargas distribuidas y a parámetros de calidad de servicio.

- Estudiar la escalabilidad de las aplicaciones distribuidas en un entorno cluster no dedicado de tamaño superior al utilizado en la experimentación desarrollada en esta tesis, de manera que permita analizar el tamaño óptimo de las aplicaciones distribuidas, en función tanto de los recursos disponibles del sistema, como al desbalanceo de la carga local y de los recursos requeridos por la propia carga distribuida.

- Analizar la viabilidad de implementar íntegramente CSC en el entorno de usuario, evaluando sus prestaciones con respecto a la versión de CSC presentada en este trabajo de tesis.

Apéndice A

Entorno de Simulación

Nuestro simulador, basado en el modelo de colas [Kle76] mostrado en la figura A.1, reproduce el modelo de cluster explicado en la sección 2.1. Este simulador ha sido implementado mediante el lenguaje de programación Java. Asimismo, para la simulación de eventos discretos se han utilizado las librerías *Desmo-J* [Des03].

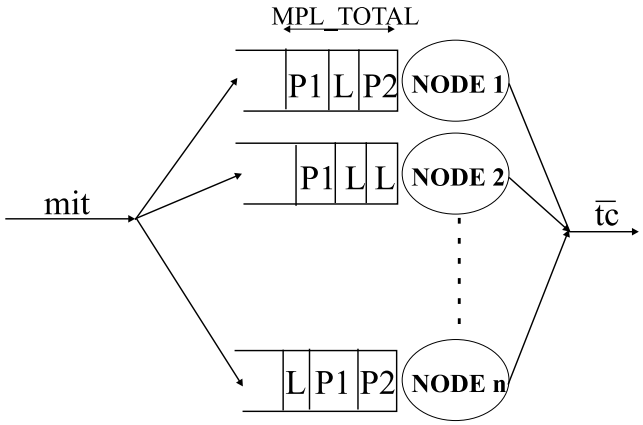

Figura A.1: Modelo de colas empleado.

Los principales parámetros de la simulación son los siguientes:

- **Tiempo de procesamiento total de una tarea** (*Job.tc*): Este valor es generado de acuerdo con una distribución *hyper-exponencial* con media $\overline{tc}$. Esta distribución modela la alta variabilidad que es esperada en entornos de supercomputación paralela [FN97]. Se asume que los trabajos generados pertenecen a dos diferentes clases: locales o paralelos; los cuales tienen asignado una media de 30000 y 360000 u.t (unidad de

tiempo), respectivamente. Cada trabajo generado es local con una probabilidad *plt* ("*probability of local task*") y distribuido con una probabilidad (*1-plt*). La función de distribución es una *Bernoully* con media *plt*. A todas las tareas que forman un determinado trabajo distribuido se les asigna un mismo tiempo de procesamiento ($Job.tc$), por lo que se asume que $Job.tc$ se corresponde con el tiempo de procesamiento en un entorno paralelo dedicado.

- **Tamaño del trabajo** ($Job.size$): El tamaño del trabajo paralelo es una potencia de dos dentro del rango $[2, ..., n]$, donde n es el número de nodos del cluster. De acuerdo con el modelo de carga definido por Feitelson [Fei96], el tamaño n ha sido escogido de acuerdo con una distribución armónica de orden 1,5. Se asume que un trabajo local está formado por una única tarea, es decir $Job.size = 1$. Un mapping estático con una tarea por nodo es asumido. El mapping de las tareas paralelas está basada en obtener una carga de memoria uniforme a lo largo del cluster. En cambio, las tareas locales son mapeadas aleatoriamente, de acuerdo con una distribución uniforme, a lo largo del cluster.

- **Tiempo medio entre arribos** (mit): Es el tiempo medio en generar un nuevo trabajo, local o paralelo, al cluster. La función escogido es una *Poisson* con media *mit*. El valor de este parámetro (para un predeterminado valor de MPL_TOTAL fijado por el usuario) es calculado de acuerdo con la siguiente expresión:

$$MPL\_TOTAL = \frac{mean\_size \times \overline{tc}}{(\overline{tc} - mit) \times n} \tag{A.1}$$

donde n es el número de nodos del cluster, $\overline{tc}$ la media de tiempo de procesamiento de un trabajo y $mean\_size$ el tamaño medio de los trabajos. Este último parámetro se calcula de acuerdo con la siguiente expresión:

$$mean\_size = \frac{\sum_k Job_k.size}{K} \tag{A.2}$$

donde K es el número de trabajos generados en el cluster.

- **Requerimientos de memoria de las tareas** ($task.vir\_mem$): Los trabajos distribuidos se clasifican en función de sus requerimientos de memoria en dos diferentes grupos: un nuevo trabajo distribuido tendrá una probabilidad *pms* de que sea grande y una probabilidad *(1-pms)* de que sea pequeño. Se asume que todas las tareas que pertenecen a un mismo trabajo distribuido tienen asignado los mismo requerimientos

de memoria. De este modo, las tareas locales o bien las tareas distribuidas pequeñas tendrán asignado un tamaño comprendido en el rango: [1,...,*2*mest*] mientras que las tareas distribuidas grandes tendrán asignado un valor comprendido en el rango:[2*mest,...,8*mest], donde *mest*=4Kpages. Una distribución uniforme ha sido escogida para asignar el tamaño de memoria (tomando como unidad el tamaño de una pagina). Inicialmente, el número de páginas residentes asociados a una tarea *task* ($task.res\_mem$) será calculado de acuerdo con la siguiente ecuación:

$$task.res\_mem = \begin{cases} task.vir\_mem & if(node_k.mem < node_k.M) \\ \frac{(task.vir\_mem) \times M}{node_k.mem} & if(node_k.mem \geq node_k.M) \end{cases} \quad \text{(A.3)}$$

donde $node_k.mem$ es la suma de los requerimientos de memoria de todas las tareas que se están ejecutando en el $node_k$ y $node_k.M$ es el tamaño de la memoria física del $node_k$. Sin embargo, durante la simulación, el tamaño residente $task.res\_mem$ será reajustado de acuerdo con el algoritmo de reemplazamiento explicado en el modelo de memoria 2.1.2.

- **Working set** ($task.wrk$): El working set [Den68, Den80] de un proceso para un instante de tiempo t y con un parámetro τ es definido como el conjunto de páginas referenciadas por dicha tarea durante las últimas τ unidades de tiempo $(t, t-\tau)$. De esta manera, el working set de un proceso es una medida de la localidad de un proceso. A principios de la década de los 70s, diferentes estudios empíricos revelaron importantes propiedades del comportamiento de memoria de los programas [Den68]. Dichos estudios mostraron que los requerimientos de memoria de los programas consistían en una serie de fases donde predominaba una elevada localidad, finalizando cada fase con una transición donde se producían la gran mayoría de fallos de página debido a la carga del conjunto de páginas que serían referenciadas durante la posterior fase. El tamaño del working set durante cada fase es simulado mediante una distribución normal con media=$0{,}5 * task_l.vir\_mem$ para las tareas locales y una media =$0{,}8 * task_l.vir\_mem$ para las tareas distribuidas, de acuerdo con los trabajos experimentales mostrados en [BHMW96]. La longitud de cada fase será calculada por medio de una distribución exponencial con media=$0,1*(task.tc)$. De este modo, un nuevo working set será generado después de cada fase.

- **Probabilidad de fallos de página** ($task.flt$): Cada vez que una tarea es planificada en la CPU, va generando referencias sucesivas a páginas pertenecientes a su working set ($task.wrk$), de acuerdo con una función

de distribución uniforme. En el caso de que la página generada no esté dentro de su conjunto residente, un nuevo fallo de página es generado e incrementado el correspondiente contador de fallos de página ($task.flt$).

- **Latencia de fallos de págin**a: Con objeto de simplificar nuestra simulación, se ha supuesto una latencia constante igual a $25u.t.$ Es decir, cada vez que la tarea genere un fallo de página, ésta es insertada en la cola de espera durante $25.u.t.$

- **Tamaño de memoria física de un nodo** ($node_k.M$): Por defecto, se toma un valor constante igual a $32K$ páginas.

- **Tiempo medio de servicio** ($task.quantum$): Es el quantum de tiempo asignado a las diferentes tareas. La función escogida es una exponencial con media $100u.t.$

- **Frecuencia de envío de mensajes** ($task.mes$): El tiempo entre los sucesivos envíos es simulado por medio de una distribución exponencial con media $freq\_com.$ El patrón de comunicación empleado es de uno a todos. Se han simulado dos tipos de primitivas de envío: una primitiva de *envío no bloqueante*, de manera que la tarea emisora reanuda su ejecución tras el envío, y una primitiva de *envío bloqueante*, de manera que la tarea emisora queda bloqueada hasta que recibe el reconocimiento de todas las tareas cooperantes a las cuales ha enviado el mensaje. La generación de una u otra primitiva ha sido implementada mediante una función de distribución *Bernoully* con media $ps.$

- **Latencia de la red**: Con objeto de simplificar nuestra simulación, se ha supuesto una latencia constante igual a $5u.t.$

Apéndice B

Resultados Experimentales Adicionales

En este apéndice se muestran aquellos resultados obtenidos, tanto por medio de la simulación como de la experimentación realizada en el cluster controlado, que nos permiten fijar todos aquellos parámetros que configuran el rendimiento de CSC. En concreto se evalua el valor de la constante *MNO*, así como los valores del *STEP* quantum y del umbral *MEM_MIN*. Asimismo, el apéndice finaliza con un análisis sobre las desviaciones obtenidas en algunas de las pruebas que se han realizado a lo largo de esta tesis.

B.1. Asignación de la Constante MNO en el CoScheduling Cooperativo

En este apartado se analiza la influencia de la constante MNO, que fija el máximo número de adelantos que puede sufrir una tarea en la cola de preparados bajo la gestión del coscheduling Cooperativo. Con este fin, la figura B.1 muestra el slowdown sufrido por las tareas distribuidas (izquierda) y locales (derecha) cuando la contante MNO es variada entre uno y tres, así como para diferentes valores de frecuencia de comunicación. Asimismo, a modo de referencia, se han mostrado los valores obtenidos con la técnica de planificación de Round-Robin (RR).

El análisis de ambas figuras muestra como un valor de $MNO = 2$ alcanza un mejor equilibrio respecto el slowdown introducido para ambos tipos de tareas. Mientras que para las distribuidas se alcanza un rendimiento cercano al mejor de los tres casos estudiados ($MNO = 3$), para las locales se alcanza un slowdown intermedio entre $MNO = 1$ ($CSC(1)$) y $MNO = 3$ ($CSC(3)$).

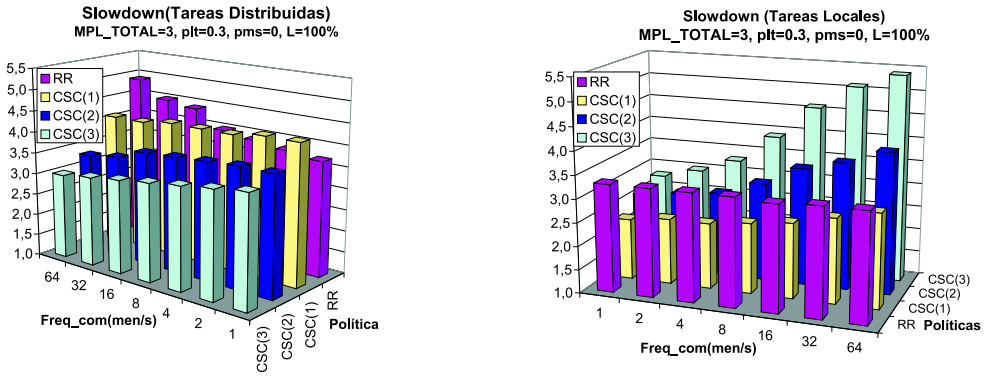

Figura B.1: Influencia de la constante MNO en el slowdown de las tareas distribuidas (izquierda) y locales (derecha).

B.2. Planificación de Trabajos con Altos Requerimientos de Memoria en el Entorno Cluster Controlado

En esta sección se evalúa en el cluster controlado, descrito en la sección 5.2, el comportamiento de CSC cuando los requerimientos de memoria superan el tamaño de la memoria principal. Esta sección se divide en dos subpartes: en la primera se evaluará el rendimiento de CSC en un entorno NOW dedicado a las tareas distribuidas, estudiando cual es la longitud de quantum que mejor se adapta a las necesidades de las tareas distribuidas en aquellos nodos donde la memoria es desbordada; mientras que en la segunda subsección se estudiará el rendimiento de CSC en entornos NOW no dedicados.

Esta experimentación ha sido realizada en el cluster *Cl1_Control*, compuesto por 8 nodos con $128MB$ de memoria principal. El porcentaje de recursos dedicados a las tareas distribuidas ha sido fijado en el *50 %*. De acuerdo con nuestro objetivo se han definido las cuatro diferentes cargas que se muestran en la tabla B.1. Las dos primeras cargas, *Wrk1* y *Wrk2*, están formadas únicamente por NAS benchmarks, siendo cada uno ejecutado con 8 tareas, mientras que en las cargas *Wrk3* y *Wrk4* se añade el benchmark *Local* con el perfil indicado entre paréntesis. Los parámetros asociados con los perfiles del benchmark Local se corresponden con los que se muestran en el apéndice E. Cabe destacar que mientras en la Wrk3 el usuario local supera la porción de memoria asignada al mismo ($= 64MB$), en la carga Wrk4 es el usuario

distribuido quien supera dicha porción. Asimismo, *la métrica mem_load,* mostrada en la tabla B.1, indica cómo está cargada, en promedio. la memoria de cada uno de los nodos del cluster. Una definición detallada de dicha métrica puede ser consultada en la sección 2.2.

Workload(Wrk)		**Estado del Cluster/mem_load**
1	FT.A+MG.B+SP.A	todos los nodos con memoria desbordada/*122 %*
2	IS.A+LU.A+FT.A	todos los nodos con memoria desbordada/*106 %*
3	MG.B+Local(Internet)	4 nodos con la memoria desbordada/*83 %*
4	FT.A+SP.A+Local(Xwin)	4 nodos con la memoria desbordada/*92 %*

Tabla B.1: Definición de las cargas de trabajo.

B.2.1. Cálculo del STEP quantum

En primer lugar se procedió a calcular, para las cargas *Wrk1* y *Wrk2*, el número de fallos de página por nodo (*FALLOS*) que se produjeron en función del *STEP* quantum. Recordemos que el módulo de memoria de CSC asigna un quantum igual a $DEF\_QUANTUM * STEP$ a aquellas tareas distribuidas residentes en aquellos nodos sin actividad de usuario local y donde la memoria ha sido desbordada (ver sección 3.4.1).

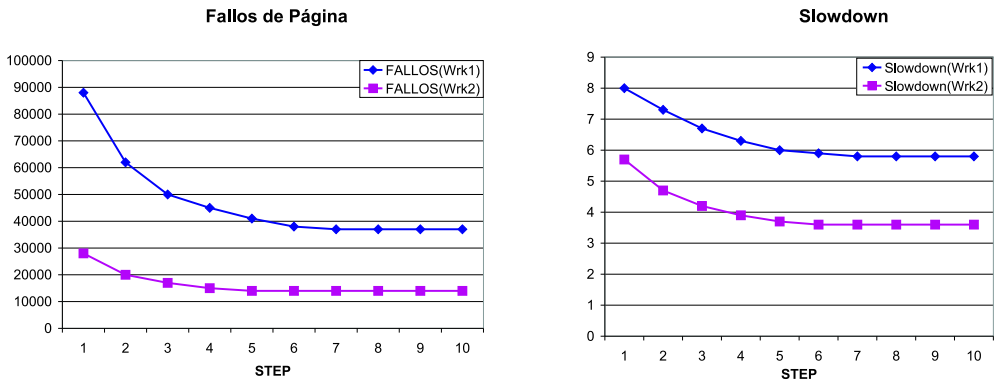

Figura B.2: Fallos de Página (izq.) y Slowdown (der.).

La figura B.2 (izq.) muestra los fallos de página y slowdown (der.) asociado con las ejecuciones de las cargas *Wrk1* y *Wrk2* al variar el STEP quantum ente 1 y 10, o lo que es lo mismo al variar la longitud del quantum entre $210ms$

y $2100ms$. El gráfico asociado con los fallos de página muestra como estos disminuyen rápidamente a medida que se incrementa el STEP, hasta llegar a un valor de $STEP = 5$, a partir del cual el número de fallos de página se mantiene estable. Estos resultados significan que ambas cargas no necesitan una rebanada de tiempo superior a $5 * 210ms$ para encajar su working set en memoria principal. Cabe destacar que esta tendencia es similar a la que se obtuvieron en las pruebas realizadas por medio de la simulación en el capítulo 2, aunque los ordenes de magnitud difieren substancialmente debido principalmente a que el tamaño del working set de las cargas utilizadas en la simulación son mucho menores que el working set asociado con las cargas reales.

Esta disminución en los fallos de página se traduce en una mejora importante del slowdown asociado con ambas cargas, alrededor del 30 %. Sin embargo, esta mejora es mayor para la carga *Wrk2*, debido a que dicha carga está compuesta por tareas con muy altos requerimientos de comunicación y sincronización, las cuales son mucho más sensibles a la disminución de los fallos de página.

Atendiendo a los resultados mostrados en esta sección se ha tomado un valor de $STEP = 5$ para el resto de pruebas realizadas en el entorno cluster experimental.

B.2.2. Rendimiento de CSC en entornos no dedicados con altos requerimientos de memoria

En esta sección se evalúa el comportamiento de CSC en aquellos nodos donde la memoria principal es desbordada, ya sea debido a que el usuario local sobrepasa la porción de memoria que éste tenía asignada, caso de la carga *Wrk3*, o bien cuando la carga distribuida sobrepasa dicha porción, caso de la carga *Wrk4*. Asimismo, se evalúa el valor que debe tomar el umbral *MEM_MIN,* el cual marca el momento de reanudación de aquella tarea distribuida detenida (stoptask) por CSC debido a una situación de excesiva paginación en un nodo. La definición de dicho umbral es descrita en detalle en la sección 3.4 del capítulo *3*.

La figura B.3 muestra la memoria residente en un nodo del cluster, asociada a cada uno de los NAS benchmarks y al benchmark *Local* que configuran las cargas *Wrk3* y *Wrk4*. Asimismo se muestra la memoria utilizada por el kernel. La figura B.3(izq.), correspondiente a la carga *Wrk3*, muestra como CSC penaliza a la tarea *Local* debido a que cumple las dos condiciones necesarias: (1) esta tarea ha sobrepasado su porción asignada de memoria y (2) se corresponde con la tarea con un mayor requerimiento de memoria. Cabe

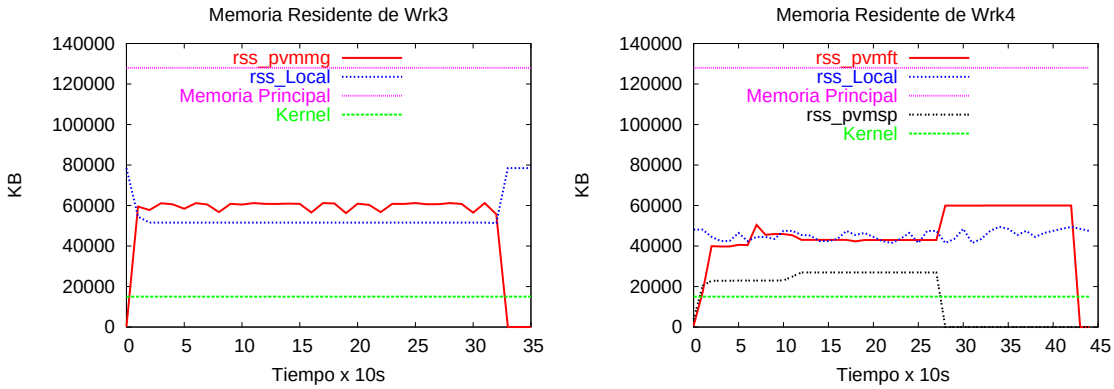

Figura B.3: Memoria residente correspondiente a la carga *Wrk3* (izq.) y *Wrk4* (der.).

decir que en el caso de que hubiese una tarea distribuida con mayores requerimientos de memoria que la tarea local, esta tarea distribuida sería la escogida para liberar páginas. Por tanto, el caso escogido se corresponde con el peor caso desde el punto de vista de los usuarios locales. No obstante, aun siendo el peor caso posible, el slowdown y la latencia de las llamadas a sistema del usuario local, ambas mostradas en la tabla B.2, se mantienen dentro de unos margenes aceptables para el usuario local.

	Tareas Distribuidas	**Tareas Locales**	
Carga	**Slowdown**	**Slowdown**	**LSC**
Wrk3	1.3	1.9	245ms

Tabla B.2: Rendimiento de las tareas distribuidas y locales con Wrk3. LSC: Latencia de las llamadas a sistema.

El comportamiento de CSC para la ejecución de *Wrk4* es diametralmente opuesto al explicado anteriormente. En este caso, CSC detiene y desaloja parcialmente de memoria en el nodo monitorizado la tarea *pvmft,* dado que es la tarea distribuida residente en dicho nodo con mayores requerimientos de memoria. Asimismo, CSC detiene al resto de tareas cooperantes de *pvmft* residentes en nodos remotos, independientemente de que la memoria de dichos nodos hubiese sido desbordada. De este modo, la tarea *pvmsp* podrá ejecutarse con todo su working set cargado en memoria y la carga Local dispondrá de todos aquellos recursos que en un principio habían sido pactados. Cuando *pvmsp* finaliza su ejecución, la tarea *pvmft* puede reanudar su

ejecución, siempre y cuando se cumpla que la suma de los requerimientos de memoria en ese nodo sea menor que la constante *MEM_MIN*. En caso afirmativo, *pvmft* será reanudada utilizando los recursos liberados por la tarea *pvmsp*.

Por tanto, un parámetro fundamental en el rendimiento del sistema CSC es el valor del umbral *MEM_MIN*. Con el objetivo de evaluar el umbral que mejor se adapta a las características del sistema, la carga *Wrk4* fue ejecutada, con la peculiaridad de que los requerimientos de memoria del usuario local variasen dinámicamente, dentro de unos margenes comprendidos entre el ± 10 % del valor medio correspondiente al perfil del usuario Local *Xwin*. Cabe decir que este intervalo fue tomado teniendo en cuenta los valores obtenidos en la monitorización realizada para el estudio del uso de recursos por parte de los usuarios de los laboratorios de la EPS de la UdL (ver apéndice E). De este modo, las fluctuaciones en los requerimientos de memoria asociados con los usuario locales son simulados.

MEM_MIN	**Aplic. Distribuidas**	**Tareas Locales**	
	Slowdown	**Slowdown**	**LSC**
0.7	∞	1	100ms
0.8	∞	1	100ms
0.9	4	1.1	110ms
0.95	1.72	1.7	185ms
0.98	1.9	1.8	195ms

Tabla B.3: Rendimiento de las aplicaciones distribuidas y locales en función del umbral *MEM_MIN*. El símbolo ∞ significa que la aplicación distribuida no es reanudada hasta que la tarea local finaliza. LSC: Latencia media de las llamadas a sistema.

La tabla B.3 muestra tanto el slowdown obtenido para las aplicaciones distribuidas, en función del valor de *MEM_MIN*, como el slowdown y la latencia de las llamadas a sistema asociadas con el usuario local. En esta tabla se observa como el slowdown de las aplicaciones distribuidas depende totalmente de este valor. Esta variación es debida a que al finalizar la tarea *pvmsp*, el resto de tareas residentes en el nodo, no sobrepasan el tamaño de la memoria pero están muy cercanos al mismo. Ésto comporta que dependiendo del valor de *MEM_MIN,* la tarea *pvmft* sea reanudada o no. Por consiguiente, en caso de que no sea alcanzado dicho umbral, la tarea *pvmft* quedará detenida hasta que el usuario local finalize su trabajo o bien disminuya de manera considerable sus requerimientos de memoria, siendo ambos casos indicados en la tabla mediante el símbolo ∞. Por otro lado, un valor excesivamente grande

de *MEM_MIN* puede ocasionar que constantemente la tarea sea detenida y reanudada dependiendo de las fluctuaciones de la memoria demandada por el usuario local. Sin embargo, el coste de esta última operación es relativamente pequeño tanto para el usuario local como para el distribuido, hecho que indica que más vale inclinarse hacía una política agresiva con un valor próximo de *MEM_MIN* al 100 % de la memoria principal. Teniendo en cuenta estas consideraciones, un valor de $MEM\_MIN = 95\,\%$ ha sido tomado a lo largo de esta tesis.

B.3. Grado de Desviación de los Resultados Experimentales

En esta sección se muestran tanto las desviaciones estándar (S) como el valor medio ($\overline{x}$) obtenidos en las diferentes pruebas realizadas en el entorno cluster controlado, mostradas en la sección 5.2.2 del capítulo 5 de experimentación. La desviación estándar se define de acuerdo con la siguiente expresión:

$$S = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \overline{x})^2}{n}} \tag{B.1}$$

donde $\overline{x}$ es la media obtenida después de realizar n pruebas.

La tabla B.4 muestra los correspondientes valores obtenidos en el cálculo del speedup medio de las tareas distribuidas, para cada una de las cargas definidas (*COM_L, COM_M* y *COM_H*) en el capítulo 5 de experimentación, al variar el grado de multiprogramación de uno a cuatro en el entorno cluster controlado. Asimismo, las tablas B.5 y B.6 muestran la media y la desviación estándar asociados con la latencia media de las llamadas a sistema (LSC) y el slowdown de las tareas locales. Estos valores se corresponden con la experimentación realizada en el cluster *Cl2_Control* con cuatro usuarios locales, cuyos valores medios son mostrados y analizados en la sección 5.2.2 del capítulo 5.

En general se observa como la evolución de la desviación mostrada en las tres tablas es muy similar. La desviación obtenida aumenta progresivamente a medida que se incrementa el grado de multiprogramación, así como a medida que se aumentan los requerimientos de comunicación de las aplicaciones distribuidas. Asimismo, estos resultados ponen de manifiesto como el coscheduling CSC obtiene unos resultados más estables que el coscheduling Dinámico (Dynamic). En este sentido cabe destacar el aumento de la desviación obtenida, sobretodo con el coscheduling Dinámico, en aquellos

casos donde la memoria es desbordada.

SpeedUp		COM_L		COM_M		COM_H	
MPL	**Política**	$\overline{x}$	**S**	$\overline{x}$	**S**	$\overline{x}$	**S**
1	*Dynamic*	5,7	0,03	5,4	0,05	4.9	0,08
	CSC	5,3	0,03	4,8	0,04	4,4	0,06
2	*Dynamic*	4,7	0,1	4,7	0,15	4,2	0,20
	CSC	4,4	0,08	4,2	0,09	3,8	0,1
3	*Dynamic*	4,1	0,15	4	0,20	1,9	0,75
	CSC	3,8	0,09	3,6	0,14	3,1	0,25
4	*Dynamic*	3,7	0,20	2,3	0,56	1,5	0,95
	CSC	3,4	0,15	3,2	0,31	2,6	0,33

Tabla B.4: Media ($\overline{x}$) y Desviación (S) asociada al cálculo del speedup de las tareas distribuidas el variar el grado de multiprogramación.

LSC		COM_L		COM_M		COM_H	
MPL	**Política**	$\overline{x}$	**S**	$\overline{x}$	**S**	$\overline{x}$	**S**
1	*Dynamic*	240msg	3msg	215msg	6msg	195msg	9msg
	CSC	190msg	3msg	145msg	3msg	135msg	2msg
2	*Dynamic*	480msg	10msg	440msg	12msg	420msg	15msg
	CSC	290msg	4msg	210msg	6msg	180msg	8msg
3	*Dynamic*	585msg	12msg	650msg	23msg	880msg	158msg
	CSC	370msg	8msg	295msg	8msg	270msg	10msg
4	*Dynamic*	798msg	24msg	1060msg	134msg	1100msg	295msg
	CSC	410msg	11msg	340msg	31msg	290msg	34msg

Tabla B.5: Media ($\overline{x}$) y Desviación (S) asociada al cálculo de la latencia media de las llamadas a sistema (LSC) al variar el grado de multiprogramación.

B.4. Correspondencia entre las Métricas del Usuario Local y los Parámetros del Sistema

Un aspecto a tener en cuenta es la correspondencia entre las métricas del usuario local, analizadas a lo largo de esta tesis, con respecto a los parámetros más representativos de la carga del sistema. Diferentes trabajos mostrados en

Slowdown		COM_L		COM_M		COM_H	
MPL	**Política**	$\overline{x}$	**S**	$\overline{x}$	**S**	$\overline{x}$	**S**
1	*Dynamic*	1,4	0,03	1,4	0,08	1,3	0,08
	CSC	1,3	0,03	1,2	0,04	1,2	0,04
2	*Dynamic*	2,1	0,10	1,9	0,14	1,9	0,15
	CSC	1,6	0,06	1,5	0,08	1,4	0,11
3	*Dynamic*	3,1	0,15	2,9	0,2	5,1	0,45
	CSC	2,3	0,09	1,9	0,12	1,7	0,15
4	*Dynamic*	4,2	0,17	4,6	0,35	5,8	0,75
	CSC	2,7	0,11	2,6	0,18	1,9	0,18

Tabla B.6: Media ($\overline{x}$) y Desviación (S) asociada al cálculo del slowdown de las tareas locales el variar el grado de multiprogramación.

la literatura [FZ87, FRS+97, BF00, ZSMF01] convergen en que los parámetros que mejor representan la carga del sistema son el porcentaje de memoria demandada y la carga de CPU, siendo la longitud de la cola de preparados el parámetro que mejor define la carga de CPU [FZ87].

	COM_L				COM_H			
	Dynamic		CSC		Dynamic		CSC	
MPL	**cpu**	**%mem**	**cpu**	**%mem**	**cpu**	**%mem**	**cpu**	**%mem**
1	0.9	35 %	0.84	35 %	0.3	55 %	0.25	55 %
2	1.85	45 %	1.81	45 %	0.45	71 %	0.40	71 %
3	2.26	53 %	2.21	53 %	1.65	88 %	0.55	88 %
4	2.85	62 %	2.74	62 %	2.25	115 %	1.3	115 %
5					2.90	130 %	1.8	130 %

Tabla B.7: Evolución de los parámetros de carga del sistema con respecto al grado de multiprogramación. *CPU*: Longitud media de la cola de preparados durante el último minuto. *%MEM*: Porcentaje medio de memoria demandada por los procesos locales y distribuidos con respecto al tamaño de la memoria principal.

Con este objetivo, en el cluster homogéneo *Cl2_Control* se midieron con la herramienta de monitorización *Monito, descrito en el apéndice* D, ambos parámetros. Los resultados mostrados en la tabla B.7 muestran la longitud media de la cola de preparados (*cpu*) y el porcentaje de memoria demandada por los procesos locales y distribuidos (*%mem*), promediados ambos en un

intervalo de un minuto. Los resultados mostrados se corresponden con la media obtenida entre los cuatro nodos del cluster ocupados por usuarios locales. En color rojo, se muestran aquellos casos dondes las métricas de usuario dieron unos valores por encima de los umbrales mínimos soportados por el usuario local (ver figura 5.6). El análisis de estos resultados muestra como en aquellas circunstancias donde toda la carga, tanto distribuida como local, encaja en memoria, caso de la carga COM_L, la política Dynamic obtiene unas métricas insatisfactorias para el usuario local con una carga de *cpu* alrededor de 1,85, mientras que CSC, debido al hecho de asignar la mitad del quantum a las tareas distribuidas, permite alargar este máximo hasta 2,2. Los resultados obtenidos con la carga COM_H muestran como, en este caso, el porcentaje de memoria se convierte en el parámetro clave que determina el rendimiento del usuario local, de modo que bajo la política Dinámica este rendimiento cae en el momento que se alcanza una demanda de memoria alrededor del 85 %. Cabe decir que aunque este porcentaje medio esté por debajo del 100 %, en numerosos nodos del cluster se produjo un desbordamiento de la memoria principal, hecho que repercutió también en un aumento importante de la carga de CPU. En esta línea, cabe destacar como CSC es capaz de rebajar dinámicamente el porcentaje de memoria demandada por los procesos activos alrededor de un 35 %, de modo que bajo CSC se alcanza un porcentaje de memoria total demandada alrededor del 125 %.

B.5. Speedup de las Aplicaciones Distribuidas en el Entorno Productivo

En esta sección se muestran los valores de speedup asociados a las aplicaciones que integran las cargas COM_L y COM_H, cuando ambas fueron ejecutadas en los tres escenarios definidos en el cluster productivo y descritos en la sección 5.3.1.

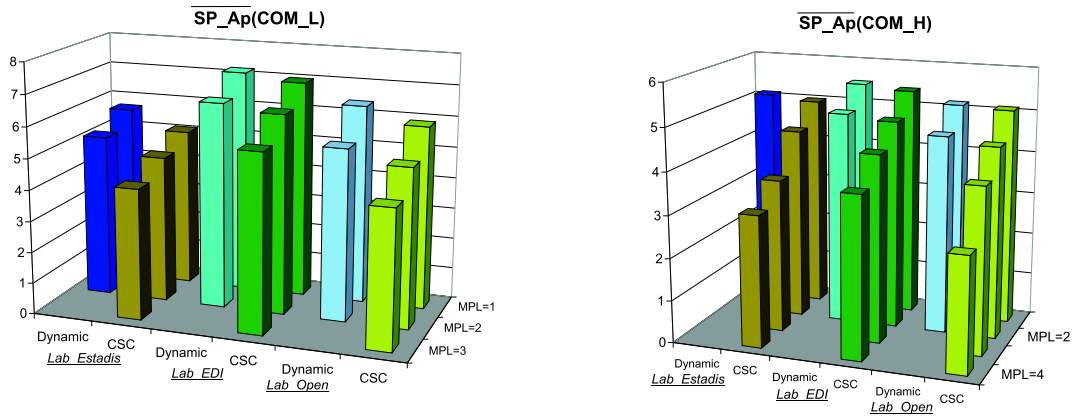

Figura B.4: Speedup a nivel de aplicación de las cargas distribuidas ejecutadas en un entorno cluster productivo.

Apéndice C

Nuevas Llamadas a Sistema Implementadas

En este apéndice se describen las cuatro llamadas a sistema implementadas en el desarrollo de *CSC*. En el primer apartado se describen las diferentes modificaciones realizadas en el kernel para su implementación. En el segundo apartado se describe el código del comando Linux asociado a las mismas.

C.1. Llamadas a Sistema

Los diferentes ficheros modificados y las nuevas lineas de códigos insertadas son las siguientes:

```
/*****************entry.S*******************/
.long SYMBOL_NAME(sys_heterogeneity) /* 191 */
.long SYMBOL_NAME(sys_view_heterogeneity) /* 192 */
.long SYMBOL_NAME(sys_coordinated) /* 193 */
.long SYMBOL_NAME(sys_view_coordinated) /*194*/
```

```
/***************unistd.h***********/
#define __NR_heterogeneity 191
#define __NR_view_heterogeneity 192
#define __NR_coordinated 193
#define __NR_view_coordinated 194
```

/**********sys.c******************/

Llamada Heterogeneity:

/*Esta llamada acepta cuatro parámetros: el % de recursos asignados a las tareas paralelas (L), los *BOGOMIPS* asociados con el procesador de mayor potencia del cluster ($BOGO_{max}$), el mínimo tamaño de memoria principal presente a lo largo del cluster, ambos utilizados para gestionar la heterogenidad del cluster, y el valor de *stepquantum (STEP)* utilizado por el bloque de asignación del quantum en aquellos nodos con altos requerimientos de memoria.

asmlinkage int sys_heterogeneity(int valor,long int valor1,int valor2,int valor3) {

int error = 0;
int divi;
percent = valor;
MEM_MIN=valor1;
BOGO_MAX=valor2;
divi=100/percent;
if(divi==1) REDUIR=0;
else if(divi<=2) REDUIR=1;
else REDUIR=2;
INCREM=BOGO_MAX/Power_Weight;
MEM_THRESHOLD=MEM_MIN*percent/400;
stepquantum=valor3;
return error; }

Llamada View_Heterogeneity:

/* Esta llamada retorna cuatro valores: la porción de memoria asignada a las tareas paralelas, el porcentaje de reducción del quantum (L), el incremento de quantum producido como consecuencia del procesamiento de la heterogeneidad del procesador y finalmente el valor de *stepquantum (STEP)* pasado por parámetro en la llamada anteriormente descrita. */

*asmlinkage int sys_view_heterogeneity(long int *valor1,int *valor2,int *valor3,int *valor4) {*

int error = 0;
*valor1= MEM_THRESHOLD;
*valor2= REDUIR;
*valor3= INCREM;
*valor4= stepquantum;
return error; }

Llamada coordinated:

/*Esta llamada es utilizada por el daemon de PVM para enviar al kernel el tipo de evento recibido y el pid de la tarea asociada con el mismo. Asimismo implementa parte del algoritmo de recepción de eventos explicado en la sección 3.5.*/

```
asmlinkage int sys_coordinated(int valor,pid_t valor1) {
struct task_struct * p;
int err = -EPERM;
event=valor;
read_lock(&tasklist_lock);
for_each_task(p) if (p->pid == valor1) goto out;
out:
if(event==0 && p->priority!=1)
        p->priority=DEF_PRIORITY>>REDUIR;
else if(event==1 && p!=taskpriotemp && p!=tasknopriotemp)
        p->priority=DEF_PRIORITY;
else if(event==2 && p!=tasknopriotemp) {
        p->priority=1;
        if(tasknopriotemp!=NULL) {
                tasknopriotemp->priority=DEF_PRIORITY;
                tasknoprio=tasknopriotemp;
                tasknopriotemp=p;
                kill_proc_info(10,1,pid_pvmd);
                DEDICATED=3;
                distri_tasks[0]=tasknoprio->pid;
                printk("Reanudem Tasca Aturada: %d\n",tasknoprio->pid);
        }
        else p->state_par=stop;
}
else if(event==3 ) {
        if(p!=tasknopriotemp && p->priority<DEF_PRIORITY)
                p->priority=DEF_PRIORITY; printk("Reanudem tasca distribuida: %d\n",p->pid);
                p->state_par=NULL;
}
read_unlock(&tasklist_lock);
return err;
}
```

Llamada view_coordinated:

/*Esta llamada es utilizada por el daemon de PVM para capturar el evento y pids de las tareas afectadas por la ocurrencia de dicho evento*/

*asmlinkage int sys_view_coordinated(int *valor1,pid_t *matriu) {*

int error=0;

*valor1=DEDICATED;

copy_to_user(matriu,distri_tasks,4*sizeof(pid_t));

return error;

}

C.2. Comandos Linux

Heterogeneitat: Este programa es utilizado para llamar a la llamada a sistema heterogenity.

```
#include <stdio.h>
#include <stdlib.h>
#include <linux/unistd.h>
_syscall4(int, heterogeneity, int, valor,int, valor1,int,valor2,int,valor4);
main(int argc, char **argv)
{
int REDUIR;
if ( argc != 5 )
{
        printf("Usage: heterogeneitat n m x y(where <n>is equal to percent resources arranged to parallel tasks, m is the minimum memory size in KB i x is the maximum BOGOMIPS across the cluster and y is the stepquantum)\n");
        exit(0);
}
heterogeneity( atoi( argv[1] ),atoi( argv[2] ) ,atoi(argv[3]),atoi(argv[4]));
printf("%d% Resources Arranged to Parallel Tasks\n",atoi(argv[1]));
printf("Minimum Memory Size=%dKB\n",atoi(argv[2]));
printf("Maximum BOGOMIPS=%d\n",atoi(argv[3]));
printf("Stepquantum=%d\n",atoi(argv[4]));
}
```

Veure_Heterogeneitat: Este programa llama a la llamada a sistema *view_heterogeneity.*

```
#include <stdio.h>
```

```
#include <stdlib.h>
#include <linux/unistd.h>
_syscall4(int, view_heterogeneity, int*, valor1,int*, valor2,int*,valor4,int*,valor5);
main(int argc, char **argv)
{
int tant;
int valor1;
int valor2;
int valor3;
int valor4;
view_heterogeneity( &valor1,&valor2,&valor3,&valor4);
printf("MEMORY THRESHOLD= %dKB\n",valor1*4);
tant=(100> >valor2);
printf("Distributed Job Resources are %d per cent \n",tant);
printf("Due to NOW herogeneity the quantum is increased = %dms\n",200*valor3);
printf("When memory is overloaded, the quantum of distributed tasks
is %d\n",valor4*200);
}
```

Apéndice D

Herramientas de Monitorización y Administración Implementadas

En este apéndice se presenta un entorno de gestión (*PVMWeb*) y monitorización (*MoniTo*) para Clusters con un DCE ("*Distributed Computing Environment*") PVM ("*Parallel Virtual Machine*") y sistema operativo Linux. Proporciona además un entorno gráfico amigable, sencillo e interactivo y facilita el acceso desde cualquier punto del Cluster, LAN (Local Area Network) o Internet.

D.1. MoniTo: Herramienta de Monitorización Implementada

En este apéndice se describe la herramienta de monitorización, denominada *MoniTo* implementada a lo largo de esta tesis. MoniTo permite monitorizar en detalle el sistema de comunicaciones, la memoria y la CPU de los diferentes nodos del cluster. La pretensión final de esta herramienta es monitorizar a bajo nivel el rendimiento del cluster con objeto de poder incidir en la búsqueda de posibles mejoras para aumentar la productividad del cluster. La realización de esta herramienta se ha realizado en dos sucesivas fases, en la primera de ellas se definió la arquitectura y se desarrolló el módulo de monitorización del sistema de comunicaciones [SGL+00], mientras que en una segunda fase, esta herramienta fue extendida con el desarrollo de la interface Web, junto con los módulos de monitorización de memoria y CPU [GSHL01b].

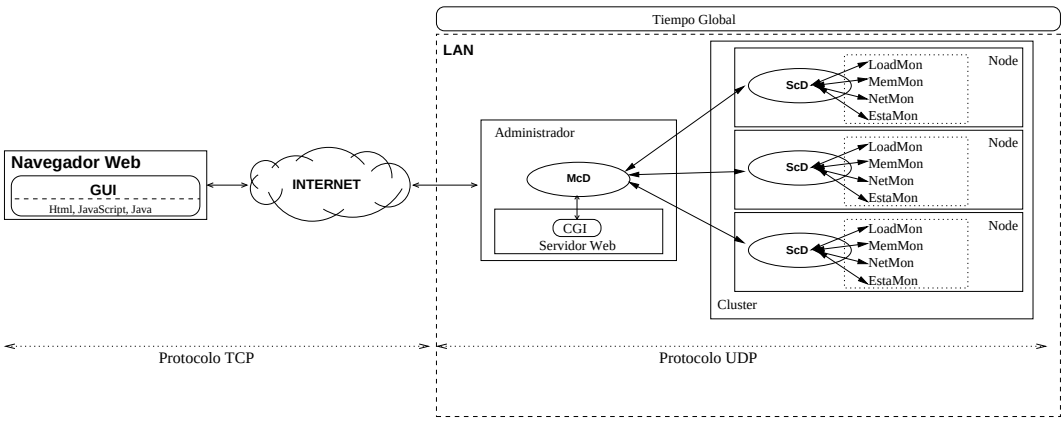

Figura D.1: Arquitectura de Monito.

D.1.1. Arquitectura

La figura D.1 muestra la arquitectura master-slave de MoniTo. En cada uno de los nodos bajo monitorización se encuentran los *ScD* (*"Slave Collector Daemon"*). Estos demonios son los encargados de recoger la información necesaria sobre el rendimiento del nodo y enviarla al *McD* ("*Master Collector Daemon*") o almacenarla en disco para un análisis posterior, según se haya seleccionado en el programa monitor. El McD, el cual se encuentra en un único ordenador del Cluster (el nodo administrador), es el encargado de recoger la información proporcionada por los ScD's a través de conexiones con el protocolo UDP[1] y enviarla, a través de una conexión con el protocolo TCP, a la herramienta de visualización. De este modo, el VcU (View and Control Unit o GUI), que es el último elemento de este sistema, presentará la información requerida en formato gráfico.

Uno de los parámetros más importantes al monitorizar es la frecuencia de muestreo. A frecuencia más alta, más precisión, pero también más intrusión (sobretodo por el incremento de tráfico en la red). De hecho, hay tres tipos de intrusión: carga de CPU, memoria y red. La manera de reducir la intrusión debida a la red es utilizar la técnica llamada *Offline*, que consiste en hacer que los datos obtenidos en cada nodo no sean enviados, si no que se almacenan en ficheros locales en cada nodo. Una vez haya finalizado la ejecución del programa paralelo, el usuario podrá analizar los resultados obtenidos en la monitorización accediendo a estos ficheros de forma directa o utilizando el programa MoniTo, que los representará de forma gráfica. Otra técnica utilizada en el programa de monitorización que también reduce el tráfico en la red,

[1]La razón de utilizar el protocolo UDP es disminuir el tráfico en la red.

es el *Buffering*. Su funcionamiento consiste en fijar un umbral de frecuencia de muestreo por encima del cual el ScD sigue capturando a la frecuencia demandada pero las envía al McD a la frecuencia umbral. Esta técnica conlleva el aumento de la intrusión de memoria. Esto implica llegar a un compromiso entre intrusión de red y de memoria.

Para sincronizar los relojes de los nodos del Cluster, donde se encuentran los ScD, con respecto el del nodo administrador y así disponer todos de la misma referencia de tiempo, se utiliza el protocolo NTP -Network Time Protocol- [Mil95]. De este modo se consigue que las muestras pertenecientes a la misma petición se tomen en el mismo instante de tiempo en cada uno de los nodos implicados.

En la figura D.1 se distingue el nodo administrador, donde se ejecuta el demonio McD y el servidor web. Este nodo será el único con acceso a Internet (para una mayor seguridad del sistema y una administración más fácil). Con objeto de garantizar una conexión entre la VcU y el McD más eficiente se utiliza el protocolo TCP (orientado a conexión).

La VcU (ver Fig. D.2) presenta al usuario los nodos disponibles para ser monitorizados y los monitores que se encuentran activos en cada uno de estos nodos. Esta unidad de visualización proporciona al usuario mecanismos para seleccionar los nodos y los parámetros a monitorizar, y también el método de monitorización.

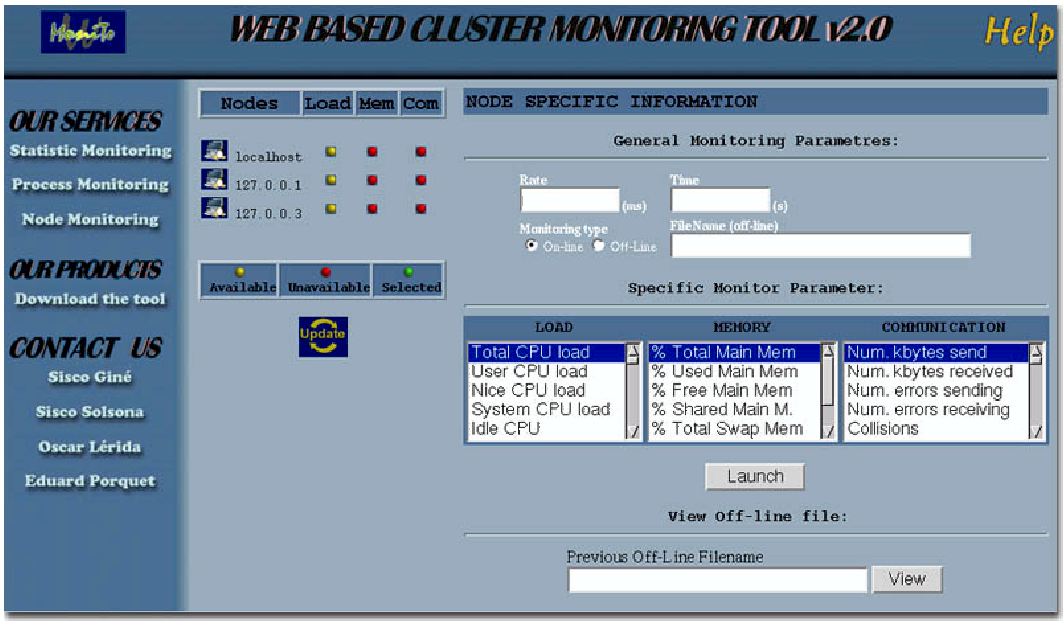

Figura D.2: Visualización gráfica de MoniTo.

D.1.2. Obtención de la información

El sistema operativo Linux ofrece dos formas de acceso a la información del sistema. Una es a través del dispositivo */dev/kmem* y otra a través del sistema de ficheros */proc*.

El acceso directo a memoria a través del dispositivo */dev/kmem* es muy rápido, pero su uso no es muy recomendable debido a que pequeñas modificaciones en versiones posteriores de Linux provocan que la herramienta de monitorización no funcione correctamente.

El uso del sistema de ficheros virtual */proc* ofrece dos grandes ventajas con respecto al anterior método, es portable e independiente con respecto a la versión de núcleo utilizado. Además, la velocidad de acceso es comparable a un acceso directo a memoria (es un sistema virtual de ficheros y el núcleo proporciona los datos directamente desde la memoria). Este es el modelo escogido en la implementación de MoniTo.

Se han diseñado tres módulos de monitorización (*stadsoc*, *stadmem* i *estadistic*), que deben ser enlazados con el núcleo en tiempo de ejecución en cada nodo, y gestionados por el demonio ScD correspondiente. Estos módulos guardan la información en unos archivos que se encuentran en el sistema de ficheros /proc. En el demonio se encuentran tres monitores (*MemMon, Netmon y EstaMon*) que acceden a la información muestreada por los tres módulos anteriores. Estos a su vez son los que controlan los archivos del sistema de ficheros /proc. Un cuarto monitor, *LoadMon*, accede también al sistema de ficheros /proc para obtener información de la carga de CPU.

A continuación se explican por separado los cuatro monitores implementados: Netmon, MemMon, LoadMon y EstaMon.

D.1.2.1. Netmon

Proporciona información específica sobre el sistema de comunicaciones, tanto de PVM como del núcleo del s.o. Linux. Para obtener información sobre las colas de transmisión y recepción de PVM (tanto de las tareas como del propio demonio), se han utilizado funciones de la librería *libpvm* de PVM.

El módulo *stadsoc* (encargado de obtener información del núcleo), introduce un nuevo archivo en el sistema de ficheros */proc/net* llamado *stadsoc*. En la tabla D.1 se muestran los parámetros de red obtenidos por stadsoc.

D.1.2.2. MemMon

Este monitor se basa en el módulo stadmem. Cuando se enlaza con el núcleo, añade un nuevo archivo en el sistema de ficheros */proc* llamado *stadmem*.

Parámetro	*Descripción*
Num. kbytes send	Número de kbytes enviados por el dispositivo de red
Num. kbytes received	Número de kbytes recibidos por el dispositivo de red
Num. errors sending	Número de errores de transmisión
Num. errors receiving	Número de errores de recepción
Collisions	Número de colisiones
Num. Buffers RecvQueue	Número de buffers en la receive_queue
Num. Buffers WriteQueue	Número de buffers en la write_queue
Num. Bytes Confirmed RQ	Número de bytes confirmados en la receive_queue
Num. Bytes Confirmed WQ	Número de bytes confirmados en la write_queue

Tabla D.1: Parámetros del monitor de comunicaciones (NetMon).

Para cada proceso y nodo a monitorizar, la información más significativa proporcionada por este monitor se muestra en la tabla D.2.

Parámetro	*Descripción*
Total Main Mem	Tamaño de la memoria principal
% Used Main Mem	Porcentaje de memoria principal utilizada
% Free Main Mem	Porcentaje de memoria principal libre
% Shared Main Mem	Porcentaje de memoria principal compartida
Total Swap Mem	Tamaño de la memoria Swap
% Used Swap Mem	Porcentaje de memoria Swap utilizada
% Free Swap Mem	Porcentaje de memoria Swap libre
Pages in swap file	Número de pàgines en el fichero de intercambio
Virtual memory size	Tamaño en páginas del espacio de direcciones virtual
Resident pages	Número de páginas residentes en memòria principal
Pages touched	Suma de maj_flt i min_flt
Pages fetch (min_flt)	Fallos de página menores (copy-on-write)
Pages fetch (maj_flt)	Fallos de página mayores
Working Set	Working set

Tabla D.2: Parámetros del monitor de memoria (MemMon).

D.1.2.3. LoadMon

Este monitor permite obtener la carga de CPU. Para ello utiliza el sistema de ficheros */proc*, y por lo tanto, no es necesario implementar ningún módulo

que genere ningún fichero. La tabla D.3 muestra los parámetro monitorizados por LoadMon.

Parámetro	*Descripción*
Total cpu load	Porcentaje de utilización de la CPU
User cpu load	Porcentaje de CPU utilizada por el usuario
System cpu load	Porcentaje de CPU utilizada por el sistema
Nice cpu load	Porcentaje de CPU utilizada por procesos con prioridad modificada
Idle cpu load	Porcentaje de CPU no utilizada
Process cpu load	Porcentaje de PU utilizada por un determinado proceso

Tabla D.3: Parámetros del monitor de carga de CPU (LoadMon).

D.1.2.4. EstaMon

Este monitor permite obtener estadísticas de uso del cluster, en relación a la carga de CPU, de memoria y de la red. El módulo que utiliza este monitor limita los intervalos de tiempo a los que un usuario tendrá acceso (último minuto, últimos cinco y quince minutos, última hora, últimas 12 horas, último día y última semana). Toda esta información se almacena en un directorio temporal como archivo de texto. La creación de este monitor surgió por la necesidad que tiene un usuario de saber cual ha sido el grado de uso de un nodo del Cluster durante un período de tiempo determinado (limitado, como hemos comentado anteriormente, a ciertos intervalos). Los parámetros que monitoriza EstaMon son: carga de CPU (total, usuario, sistema, procesos con prioridad modificada, y no utilizada), tamaño de Memoria Principal, porcentajes de MP (utilización, libre, compartida), tamaño de *swap*, porcentajes de *swap* (utilizada, libre) y KBytes enviados/recibidos, errores de transmisión y recepción y colisiones del dispositivo de red. La figura D.3 muestra la interficie utilizada por el usuario para seleccionar dichos parámetros.

D.1.3. Resultados

El overhead introducido por *MoniTo* se ha medido sobre un cluster compuesto por 4 Pentium II a 350 MHz y una memoria principal de 128 MB. Como nodo administrador hemos utilizado un Pentium III a 800 MHz y 256 MB de memoria principal. Como red de interconexión hemos utilizado una Fast Ethernet de 100Mbps.

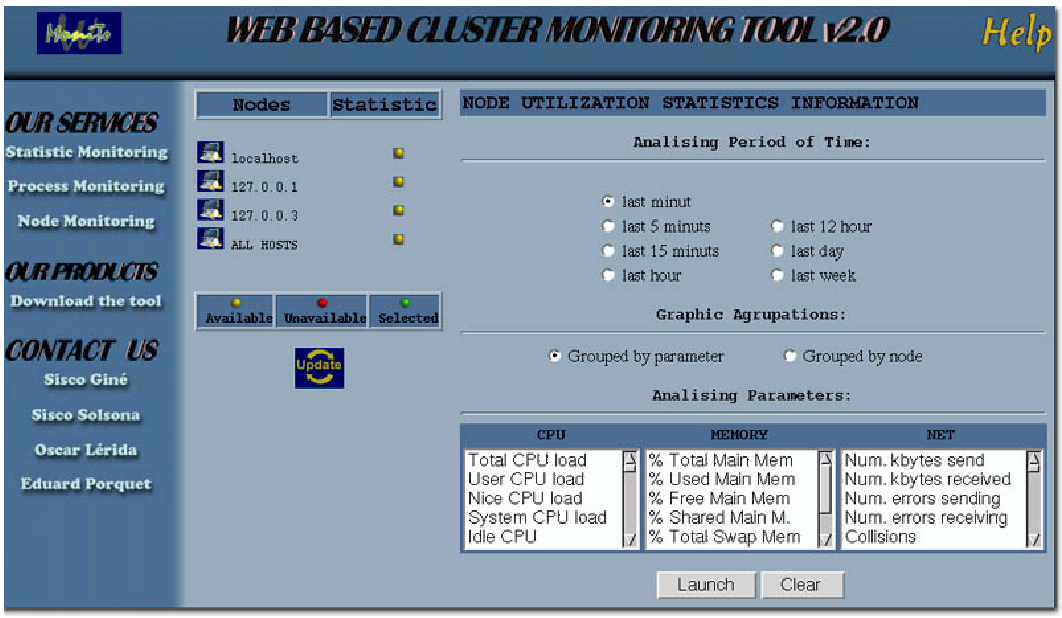

Figura D.3: Ventana de configuración del módulo estadístico.

El overhead introducido por Monito se ha obtenido en tres diferentes entornos: (a) utilizando MoniTo sin ningún método de reducción de intrusión (*Original*), así como utilizando tanto las técnicas de (b) *Buffering* como las de (c) *Offline*. El overhead es el factor en que aumenta el tiempo de ejecución (en %) de un programa paralelo cuando es monitorizado. En concreto ha sido calculado de acuerdo con la siguiente expresión:

$$Overhead = \left(1 - \frac{Tn}{Ts}\right) * 100,$$

donde Tn es el tiempo normal de ejecución del programa y Ts es el tiempo de ejecución del programa con monitorización. Los benchmarks paralelos utilizados son del NAS suite: El MG (algoritmo multi-malla) y el IS (algoritmo de ordenación de enteros).

Los resultados obtenidos (ver Figura D.4) nos permiten confirmar que las técnicas de Buffering y Offline funcionan perfectamente acorde a lo esperado. Como puede verse en esta figura, el overhead introducido por *MoniTo* es bastante aceptable, especialmente en los casos de *Buffering* i *Offline*.

Podemos concluir que *MoniTo* puede ser de gran utilidad para obtener información acerca de las aplicaciones monitorizadas sin introducir apenas overhead adicional.

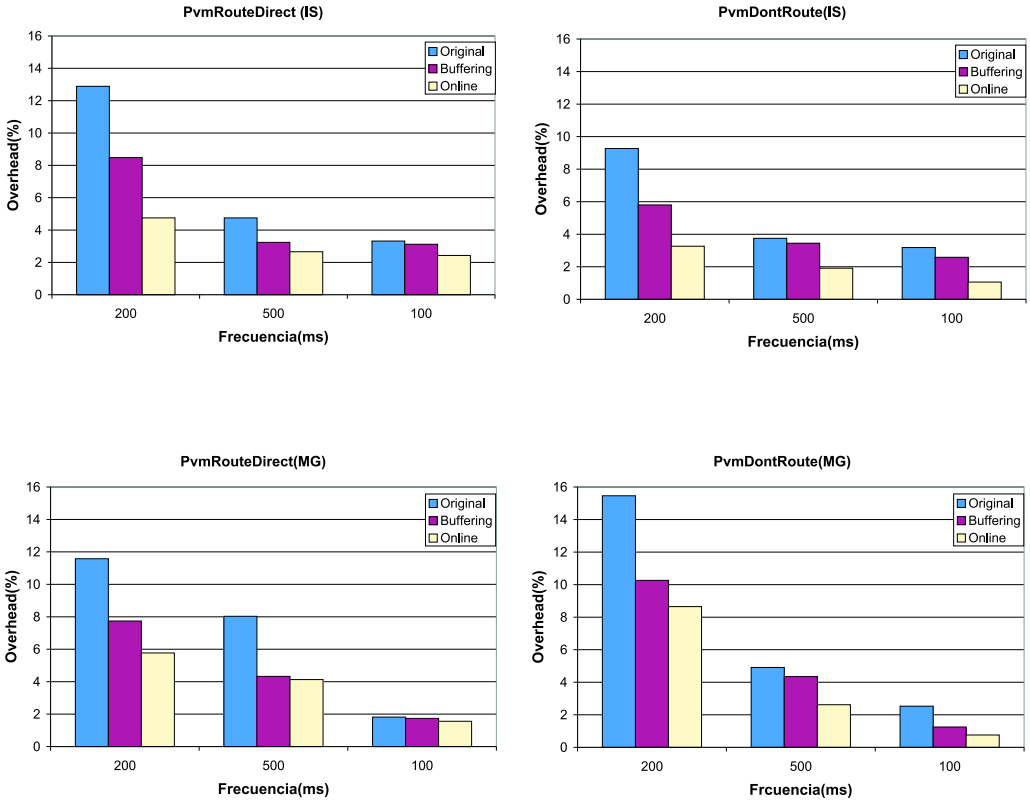

Figura D.4: Comparativa del *overhead* introducido por las diferentes técnicas implementadas.

D.2. PVMWeb: Consola de Ejecución de CSC

La ejecución de múltiples programas paralelos en un entorno no dedicado conlleva la necesidad, de dotar al usuario paralelo, de una herramienta que le permita lanzar y gestionar sus trabajos paralelos de un modo remoto, de manera que el usuario paralelo no tenga la necesidad de tener que desplazarse al laboratorio donde éstas son ejecutadas. Por otro lado, la instalación y administración de CSC requiere que cada nodo del cluster sea sintonizado individualmente por medio de las diferentes llamadas a sistema implementadas. De este modo, el administrador de CSC necesita una herramienta que le facilite la administración del sistema remotamente, de modo que desde una única consola se pueda administrar el funcionamiento de toda la Maquina Virtual Paralela (MVP: nodos activos del cluster).

Por este motivo, se ha desarrollado una herramienta, denominada *PVM-*

Web [iB03], que permite la ejecución desde cualquier punto del cluster, LAN (*Local Area Network*) o Internet, de aplicaciones paralelas PVM; así como la administración y sintonización del entorno cluster, en general, y de la técnica de coscheduling subyacente, en particular. PVMWeb ha sido implementado mediante el lenguaje Java [Jav], de acuerdo con un modelo de aplicación con *applets*. Un applet es un programa autónomo descargable desde un servidor Web, el cual se puede visualizar en un navegador.

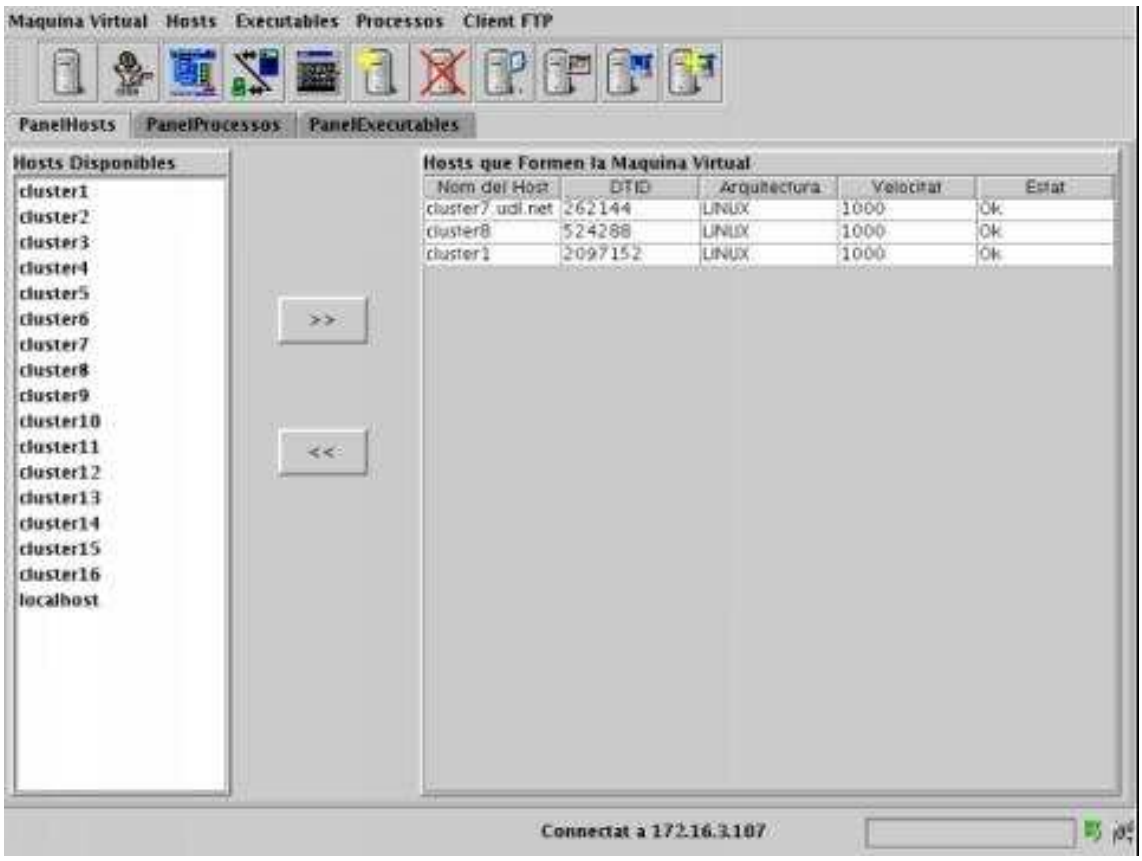

Figura D.5: Visualización gráfica de PVMWeb.

A continuación se detallan, por separado, las principales funcionalidades de PVMWeb (ver figura D.5):

- **Consola PVM**: Implementa la mayoría de comandos de la consola original de PVM (ver tabla D.4). Los comandos PVM no mostrados en dicha tabla no han sido implementados porque su ámbito de utilidad se limita a la consola de PVM (p.e. alias, unalias).

Comandos Implementados			
add	halt	ps	setenv
conf	jobs	pstat	setopt
delete	kill	put	sig
export	mstat	quit	spawn
getopt	names	reset	unexport

Tabla D.4: Lista de comandos de la consola PVM implementados en el entorno PVMWeb.

- **Cliente FTP**. Permite realizar conexiones FTP a nodos, fuera y dentro de la MVP.
- **Ejecución remota de comandos**. Permite la ejecución remota de comandos del sistema (i.e. Linux o CSC)
- **Variables de entorno**. Permite cambiar el valor de las variables de entorno (o definir otras nuevas) en un conjunto de nodos de la MVP.
- **Gestión de ejecutables**. Permite copiar ejecutables entre nodos o iniciar un ejecutable que reside en un solo nodo de la MVP. También es posible definir filtros para listar ciertos ficheros ejecutables de un conjunto de nodos de la MVP. Se puede filtrar según la combinación de nombre del nodo, nombre del ejecutable o usuario propietario del ejecutable.
- **Información de procesos**. Para cada proceso se visualiza su identificador PVM, su correspondiente identificador Linux, así como el nodo en que reside. Dicha información se puede representar de dos formas distintas: en forma de lista o en forma de árbol genealógico. Otras funciones adicionales son la eliminación o señalización de procesos de forma interactiva. Asimismo, se puede consultar el estado de cada uno de los procesos.
- **Estado de la MVP**: Permite consultar el estado de cada uno de los nodos que integran la MVP.

D.2.1. Arquitectura de PVMWeb

La arquitectura de *PVMWeb* es del tipo Cliente-Servidor (ver figura D.6). La aplicación consta de dos partes bien diferenciadas: un *Cliente*, el cual re-

alizará peticiones de servicio, y un *Servidor,* encargado de atender y responder a dichas peticiones.

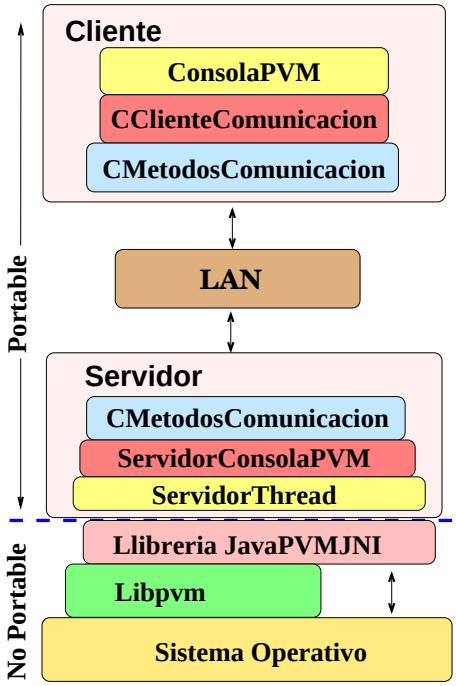

Figura D.6: Arquitectura cliente-servidor.

El Cliente está formado por 3 clases: ConsolaPVM, CClienteComunicación y CMétodosComunicación. En la *ConsolaPVM* reside toda la parte visual del cliente (la interface con el usuario). Su independencia con del resto de clases hace que sea fácilmente modificable para futuras ampliaciones. Básicamente, esta clase permite visualizar la información proporcionada por las clases de nivel inferior (CClienteComunicación y CMétodosComunicación). La clase *CClienteComunicación* contiene métodos y funciones de comunicación con el servidor, mientras que la clase *CMétodosComunicación* es la responsable de la comunicación a través de la LAN. Esta clase también es utilizada por el servidor.

El Servidor está formado también por tres clases: ServidorConsolaPVM, ServidorThread y CMétodosComunicación. *ServidorConsolaPVM* es la parte responsable de esperar nuevas conexiones del cliente. En respuesta de petición de conexión por parte del Cliente, *ServidorConsolaPVM* activa la clase *ServidorThread*, encargada de crear un thread, el cual servirá la petición del Cliente.

PVM, como se comentó en la sección 4.1.5, realiza la gestión y el control de la MVP mediante la librería *libpvm*. Debido a que Java no permite utilizar librerías en código nativo (en el caso que nos ocupa C) a libpvm, se ha realizado una librería puente entre el servidor y libpvm, denominada *JavaPVMJNI*, implementada mediante JNI [Jav] ("*Java Native Interface*"). JNI proporciona reglas y métodos para realizar llamadas a lenguajes nativos desde programas realizados en Java.

Una característica a destacar de JavaPVMJNI es que ha sido pensada para poderse ampliar fácilmente. Además es la parte que implementa la interface con el DCE (*"Distributed Computing Environment"*), en este caso PVM. Sería fácil pues incorporar nuevas funcionalidades a esta clase para que soportara, por ejemplo, otros DCEs.

Apéndice E

Caracterización de los Usuarios Locales

En este apéndice se describe cómo se ha implementado el benchmark sintético, denominado *Local*, que simula los requerimientos de un usuario local.

E.1. Implementación de *Local*

Este benchmark permite caracterizar los requerimientos de carga, por parte de un usuario genérico, de los siguientes cuatro sistemas:

- CPU
- Memoria
- Entorno de Red
- E/S a disco

En las siguientes subsecciones se describen los algoritmos empleados para la simulación de la carga asociada a cada uno de los sistemas mencionados.

E.1.1. Simulación de la carga de CPU

Para la simulación de la carga de CPU se ha implementado un algoritmo que genera una carga de CPU hasta los límites fijados por el usuario, de modo que una vez alcanzada la carga deseada, ésta se mantiene durante un tiempo fijado por el usuario.

El primer problema que se identificó era que la generación de la carga tenía que ser independiente de la potencia del procesador, es decir, si se quería incrementar la carga hasta 0.5 se tenía que generar un proceso que intercalara tiempos de CPU intensivos con tiempos de CPU que mantuviesen esa carga, lo que comportaba generar un algoritmo que creara un cómputo relativo a la potencia del procesador. Una primera idea podría basarse en la frecuencia del microprocesador, pero este parámetro nos daría un cálculo erróneo ya que muchas arquitecturas no solo dependen de la frecuencia de su microprocesador, si no de muchos otros parámetros como buses internos o cachés de primer o segundo nivel.

Descartada la frecuencia del microprocesador se consideró que la mejor opción era un parámetro aportado por el propio S.O. Linux, denominado *BogoMIPS*. Este parámetro, calculado por el núcleo de Linux durante la secuencia de arranque, mide cuanto de rápido se ejecuta una determinada rutina. El valor obtenido puede ser accedido desde el espacio de usuario mediante la lectura del fichero */proc/cpuinfo*.

Con objeto de generar una carga determinada en el sistema se optó por realizar un programa que generara los números primos por debajo de un número n. Este número está ponderado en función de los bogoMIPS de la máquina donde se ejecute el programa, de modo que en cualquier máquina que se ejecute nos de un proceso que incremente la carga de CPU. La implementación del programa se basó en el algoritmo de *Sieve Eratosthenes* [Jai91] para la búsqueda de números primos.

Con objeto de conocer en tiempo real la carga del sistema, se optó por hacer un acceso al fichero */proc/loadavg* que, en un sistema Linux con kernel 2.2.15, tiene la siguiente estructura

```
% cat /proc/loadavg
0.50 0.16 0.05 2/60 1885;
```

donde a nosotros solo nos interesan los tres primeros campos que corresponden a la carga media del sistema en los últimos 1, 5 y 15 minutos, respectivamente. En concreto, esta carga indica la media de número de procesos en la cola de preparados.

Una vez obtenidos todos los parámetros necesarios, el algoritmo implementado consiste en evaluar la carga actual del sistema, de modo que si es inferior a la carga deseada se genera un proceso que cargará la CPU usando el algoritmo de *sieves*; en caso contrario, es decir cuando la carga leída de */proc/loadavg* sea igual o superior a la carga deseada el proceso de carga entra en espera hasta que la carga deseada, sea inferior a la carga actual. En la figura E.1 se muestra el algoritmo descrito.

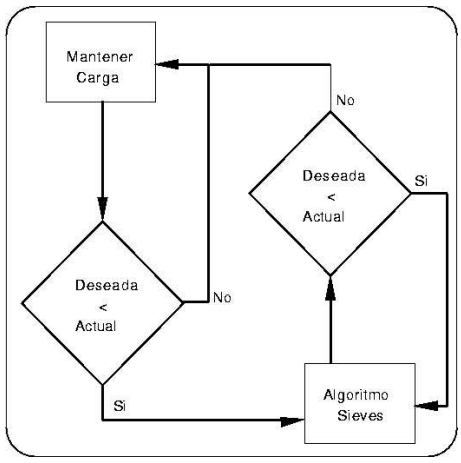

Figura E.1: Algoritmo de carga de CPU.

E.1.2. Simulación de la carga de memoria

La simulación de una carga aplicada a la memoria RAM, implica la reserva y posterior utilización de un determinado porcentaje, fijado por el usuario, de la memoria RAM. Con este objetivo, se ha creado un programa que recibe el porcentaje de memoria RAM que se quiere reservar y después de hacer un acceso a */proc/meminfo,* para conocer el tamaño de memoria principal de la máquina, reserva la cantidad de memoria asociada; recorriendo posteriormente toda la memoria reservada para que quede marcada como utilizada.

E.1.3. Simulación de la carga de comunicación

En un principio se procuró que la simulación se pudiera ejecutar localmente y sin la necesidad de usar subsistemas externos al entorno local simulado. Sin embargo, en la simulación del entorno de red se necesitaba de otro terminal remoto que enviara datos para simular un comportamiento de red. De este modo, el módulo que simula la recepción de datos tiene que ser ejecutado desde una maquina remota pasándole como parámetro la maquina donde se realiza la simulación. Asimismo, se ha implementado un módulo local que simula la transmisión de datos.

La simulación del envío de datos se realiza enviando mensajes en *Broadcast,* desde el terminal donde se está haciendo la simulación, con un tamaño y una velocidad de emisión pasados por parámetro. Para la obtención de la IP de *broadcast,* se implementó un script que usando la llamada a sistema *ifconfig* obtuviera la IP de *broadcast.* El script implementado es el siguiente:

```
% cat script_ip
#!/bin/sh
/sbin/ifconfig eth0 | grep "Bcast"
| awk '{print $3 }' | awk -F: '{print $2}'
```

Para la simulación de la recepción de datos se implementó un módulo que tiene que ser ejecutado remótamente, es decir, al módulo de simulación se le pasará como parámetro la IP de la máquina sobre la cual se está realizando la simulación. Básicamente, este módulo remoto de datos envía paquetes, hacia la maquina local, de un tamaño y una velocidad de transmisión (paquetes/segundo) indicado desde la línea de comandos.

E.1.4. Simulación de la E/S a disco

El algoritmo para la simulación de la escritura a disco crea un fichero de texto en el dispositivo físico y va aumentando su tamaño hasta alcanzar el límite de Kbytes deseados por el usuario. El algoritmo de lectura a disco realiza una lectura secuencial de los Kb indicados por el usuario. Ambas operaciones se realizan periodicamente de acuerdo con la frecuencia indicada por el usuario desde la línea de comandos.

E.2. Perfiles de Usuario Local

Con objeto de fijar los valores de las cuatro cargas simuladas de la forma más realista posible se procedio a monitorizar , mediante la herramienta de monitorización *Sar* [Sar], 10 puestos de trabajo correspondientes a un laboratorio de usuarios de la Escuela Universitaria Politécnica de la Universitat de Lleida. Esta monitorización fue realizada durante 10 dias laborables consecutivos del mes de Noviembre, las 24 horas del día.

La monitorización realizada mostró que, en general, los usuarios locales pueden agruparse en tres diferentes perfiles de usuario:

- **Xwindows**: Un *62 %* de usuarios se corresponden a este perfil. Este perfil agrupa todos aquellos usuarios que realiza su trabajo en un entorno de ventanas (*Gnome, KDE*) pero que no hace un uso exhaustivo de la red. Este usuario se dedica básicamente a tareas de edición, mediante el uso de editores de texto (*emacs, Kwrite*), o bien a tareas de edición grafica, mediante el uso de herramientas como pueden ser *xfig* o *gnuplot*.

- **Internet**: Este es el típico usuario que además de trabajar en un entorno gráfico, se dedica principalmente a realizar un uso intensivo de la red. Se dedicará principalmente al uso de navegadores web (navegación, consulta de mail, descarga de algún fichero), pero también sacará partido a las utilidades que le proporciona el entorno de ventanas. Un *33 %* de usuarios se agrupan en este perfil.

- **Shell**: El resto de usuarios son aquellos que no hacen uso del entorno de ventanas, realizando todos sus trabajos en los terminales de Linux. Generalmente es un usuario que se dedicará principalmente a tareas de compilación, edición y ejecución.

La tabla E.1 muestra los valores obtenidos en la monitorización realizada, agrupados de acuerdo con los perfiles anteriormente descritos.

Local	*CPU*	*%Memoria*	*E/S a disco (blocs/s)* *Leídos - escritos*	*Red (Bytes/s)* *Rec. - Env.*
Shell	0.25 (0.2)	20 % (15 %)	13 - 12 (4-3)	108 - 3 (30-1)
Xwin	0.15 (0.1)	35 % (55 %)	67 - 17 (23-12)	608 - 30 (302-5)
Internet	0.2 (0.1)	60 % (75 %)	99 - 52 (54-21)	3154 - 496 (1890-245)

Tabla E.1: Requerimientos medios de los usuarios locales. En paréntesis se muestran las desviaciones estandar asociadas a cada valor.

E.3. Métricas Retornadas por el Benchmark Local

Con objeto de conocer el grado de intrusión introducido por la ejecución de las aplicaciones distribuidas en las tareas locales, este mismo benchmark retorna dos diferentes métricas:

- El tiempo de ejecución en realizar un determinado número de iteraciones del algoritmo de *Sieve Eratosthenes,* utilizado para la simulación de la carga de CPU, descrito anteriormente.

- La latencia media en realizar 20 diferentes llamadas a sistema. Estas llamadas, extraidas de la suite de benchmarks *Lmbench* [VS97], son

representativas de diferentes acciones realizadas por un usuario interactivo, como por ejemplo, lecturas y escrituras a disco, envío y recepción de paquetes, creación y eliminación de procesos, creación y eliminación de ficheros, etc..

E.4. Validación Experimental de la simulación Realizada

Las pruebas se realizaron con el fin de evaluar el impacto producido por un usuario local en la ejecución de una aplicación distribuida. Se realizaron distintas experimentaciones simulando uno, dos y tres usuarios en un entorno cluster formado por ocho estaciones de trabajo. En cada prueba se comparó el impacto producido por el benchmark *Local* implementado, con respecto al impacto producido por usuarios reales comportandose de acuerdo con el perfil simulado. Los benchmarks distribuidos que se usaron en la experimentación fueron los siguientes programas de la NAS: IS.A, MG.A, BT.B y FT.B. Cada prueba fue repetida tres veces, de modo que los resultados mostrados corresponden a la media obtenida.

E.4.1. Usuario Xwindows

En las gráficas E.2, E.3, E.4 y E.5, se pueden comparar los tiempos de ejecución de los benchmarks distribuidos obtenidos cuando en el sistema había uno, dos o tres usuarios locales reales con perfil *Xwindows* o bien cuando estos eran simulados.

En las figuras E.2 y E.3 se observa como la simulación es bastante precisa consiguiendo una desviación mínima. En cambio, en los benchmarks FT y MG (ver figuras E.4 y E.5) se observa como los tiempos no son tan precisos, sobre todo en el benchmark FT.B, debido a que este benchmark hace un uso bastante elevado de la memoria y cuando empieza a utilizar memoria swap nos dan unos resultados que difieren de la simulación realizada. Asimismo, se observa como esta diferencia se agrava con el número de usuarios.

E.4.2. Usuario Internet

En las experimentaciones realizadas con un usuario tipo *Internet* se obtuvieron unos resultados similares a los obtenidos con el usuario *Xwindows.* Como se puede ver en las figuras E.6 y E.7, la ejecución del IS y del BT muestra una mínima desviación. En las figuras E.8 y E.9 podemos observar los tiempos obtenidos con los otros dos benchmarks ejecutados, el FT y

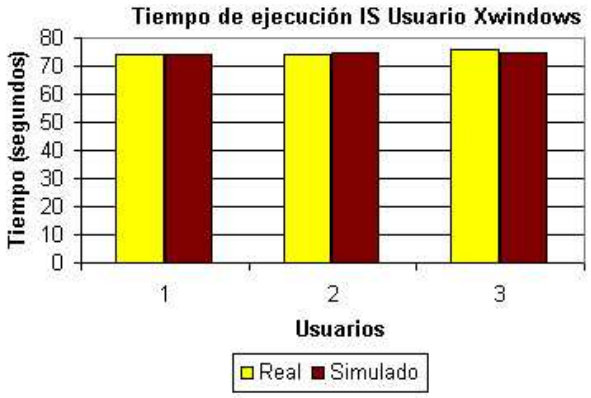

Figura E.2: Benchmark IS.A Usuario Xwindows

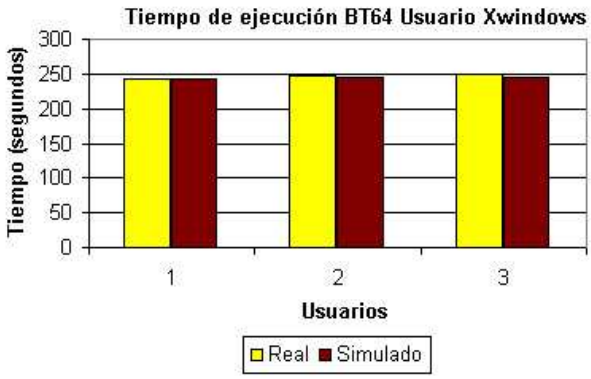

Figura E.3: Benchmark BT.B Usuario Xwindows

MG. En el FT podemos observar unas ligeras desviaciones debido a la misma causa expuesta en el punto anterior, agravada por el uso intensivo de memoria RAM que realiza este tipo de usuario local. En el benchmark MG podemos observar un buen comportamiento en la simulación de uno, dos y tres usuarios con ligeras desviaciones.

E.4.3. Usuario Shell

En las figuras E.11, E.12, E.10 y E.13 se puede observar como la simulación implementada se comporta prácticamente igual que los usuarios simulados. Estos resultados confirman que el subsistema más crítico a simular es el comportamiento de la memoria.

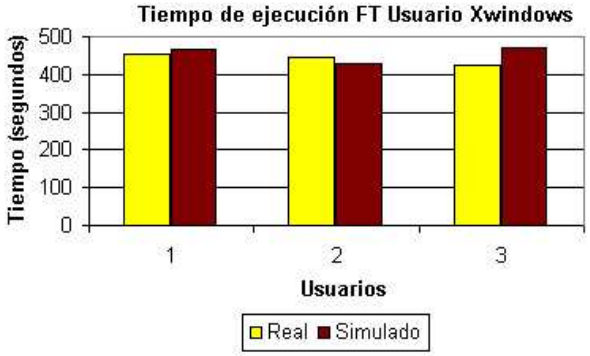

Figura E.4: Benchmark FT.B Usuario Xwindows

Figura E.5: Benchmark MG.A Usuario Xwindows

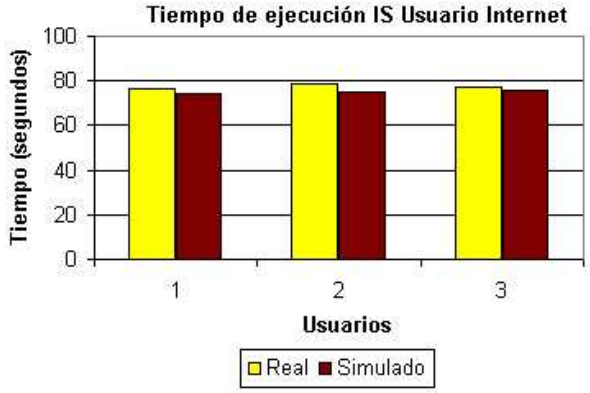

Figura E.6: Benchmark IS.A Usuario Internet.

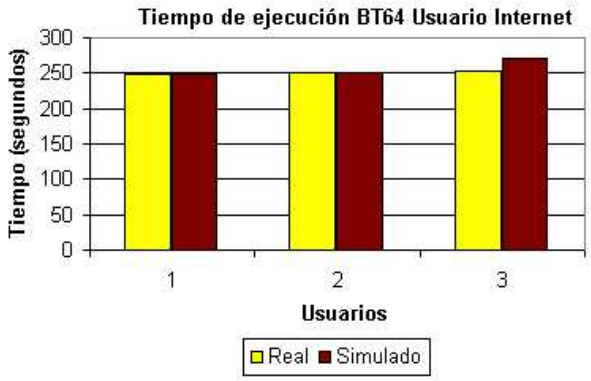

Figura E.7: Benchmark BT.B Usuario Internet.

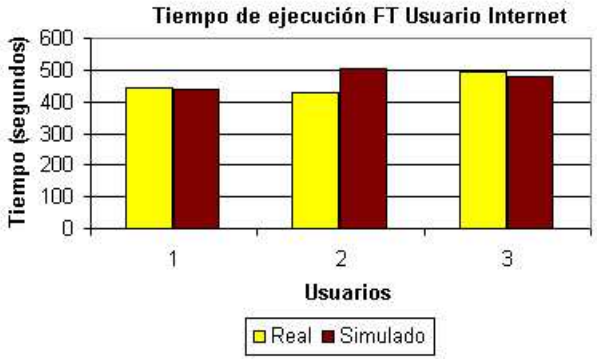

Figura E.8: Benchmark FT.B Usuario Internet.

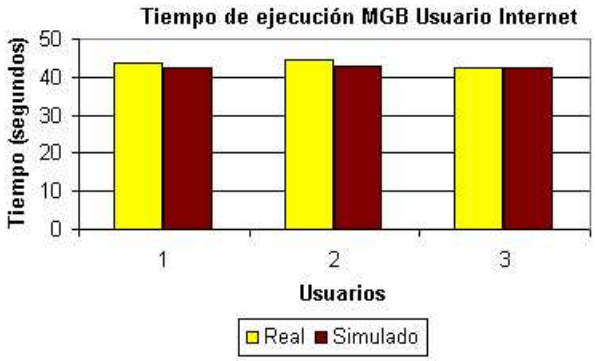

Figura E.9: Benchmark MG.A Usuario Internet.

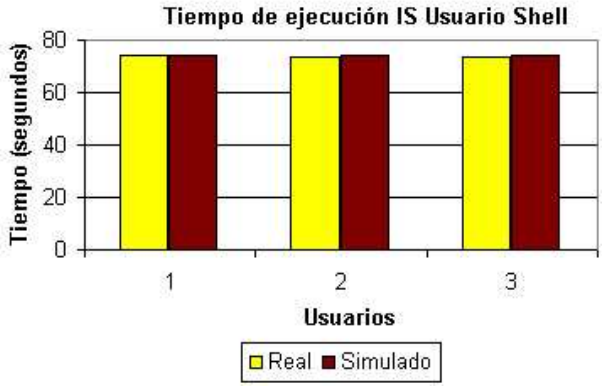

Figura E.10: Benchmark IS.A Usuario Shell

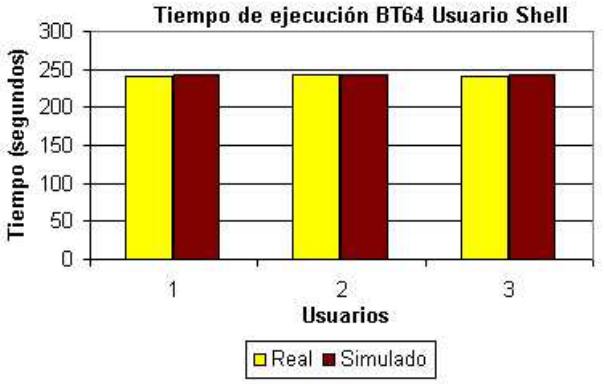

Figura E.11: Benchmark BT.B Usuario Shell

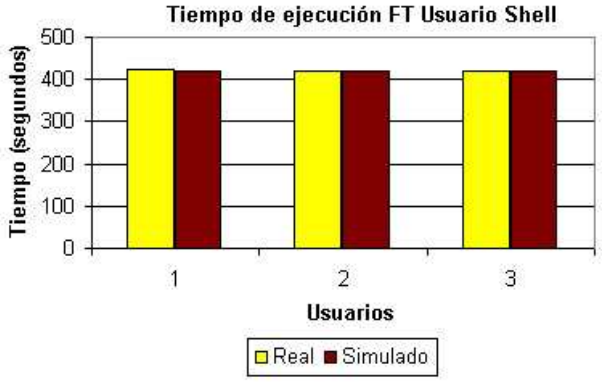

Figura E.12: Benchmark FT.B Usuario Shell

Figura E.13: Benchmark MG.A Usuario Shell

Bibliografía

[ACP+95] T. Anderson, D. Culler, D. Paterson, et al. A case for network of workstations: Now. *IEEE Micro*, 15(1):54–64, 1995.

[AD01] A. Arpaci-Dusseau. Implicit coscheduling: coordinated scheduling with implicit information in distributed systems. *ACM Transactions on Computer Systems*, 19(3):283–331, 2001.

[ADC97] A. Arpaci-Dusseau and D. Culler. Extending proportional-share scheduling to a network of workstations. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'97)*, 1997.

[ADCM98] A. Arpaci-Dusseau, D. Culler, and A. Mainwaring. Scheduling with implicit information in distributed systems. In *Proceedings of the Measurement and Modeling of Computer Systems Conference*, pages 233–243, 1998.

[ADV+95] R. Arpaci, A. Dusseau, A. Vahdat, L. Liu, T. Anderson, and D. Patterson. The interaction of parallel and sequential workloads on a network of workstations. In *Proceedings of ACM SIGMETRICS/PERFORMANCE'95*, pages 267–278, 1995.

[AES97] A. Acharya, G. Edjlali, and J. Saltz. The utility of exploiting idle workstations for parallel computations. In *Proceedings of the ACM SIGMETRICS/PERFORMANCE'97*, pages 225–236, 1997.

[Aid00] K. Aida. Effect of job size characteristics on job scheduling performance. In *Job Scheduling Strategies for Parallel Processing*, volume 1911 of *Lecture Notes in Computer Science*, pages 1–10. 2000.

[All96] Gigabit Ethernet Alliance. *Gigabit Ethernet:White Paper.* http://www.gigabit_ethernet.org/technology/whitepapers/gige, 1996.

[Ang00] C. Anglano. A comparative evaluation of implicit coscheduling strategies for networks of workstations. In *Proceedings of the 9th International Symposium on High Performance Distributed Computing (HPDC'2000)*, pages 221–228, 2000.

[AS99] A. Acharya and S. Setia. Availability and utility of idle memory in workstation clusters. In *Proceedings of the ACM SIGMETRICS/PERFORMANCE'99*, pages 35–46, 1999.

[Bac86] M. Bach. *The Design of the UNIX Operating System.* Prentice-Hall International Pub., 1986.

[Bai02] D. Llobera Baiget. Influencia de la longitud del quantum en el rendimiento de la memoria cache. Trabajo fin de carrera, Dep. Informática, Universitat de Lleida, 2002.

[BBB+94] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, , P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. The nas parallel benchmarks. Tech. report rnr-94-007, NASA Ames Research Center, 1994.

[BC01] D. Bovet and M. Cesati. *Understanding the Linux Kernel.* O'Reilly & Associates Pub., 2001.

[BCS93] E. Biagion, E. Cooper, and R. Sansom. Designing a practical atm lan. *IEEE Network*, pages 32–39, 1993.

[BF00] A. Batat and D. Feitelson. Gang scheduling with memory considerations. In *Proceeding of the 14th Intl. Parallel Distributed Processing*, pages 109–114, 2000.

[BHMW96] D. Burger, R. Hyder, B. Miller, and D. Wood. Paging tradeoffs in distributed shared-memory multiprocessors. *J. of Supercomputing*, 10(1):87–104, 1996.

[BL98] A. Barak and O. La'adan. The mosix multicomputer operating system for high performance cluster computing. *J. of Future Generation Computer Systems*, 1998.

[Buy99] R. Buyya, editor. *High Performance Cluster Computing*, volume 1. Prentice Hall PTR Pub., 1999.

[CADG+93] D. Culler, A. Arpaci-Dusseau, S. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel programming in split-c. In *Proceedings of Supercomputing'93*, pages 262–273, 1993.

[CAK+03] G. Choi, S. Agarwal, J. Kim, A. Yoo, and C. Das. Impact of job allocation strategies on communication-driven coscheduling in clusters. In *Euro-Par'2003 Parallel Processing*, volume 2790 of *Lecture Notes in Computer Science*, pages 160–169. 2003.

[CDD+94] M. Crovella, P. Das, C. Dubnicki, T. Leblanc, and E. Markatos. Multiprogramming on multiprocessors. In *Proceedings of the 3rd. IEEE Symposium on Parallel and Distributed Processing*, pages 590–597, 1994.

[CFGK95] N. Carriero, E. Freedman, D. Gelernter, and D. Kaminsky. Adaptive parallelism and piranha. *Computer*, 28(1):40–49, 1995.

[CNSW97] A. Chowdary, L. Nicklas, S. Setia, and E. White. Supporting dynamic space-sharing on clusters of non-dedicated workstations. In *Proceedings of 17th Intl. Conference on Distributed Computing Systems*, pages 149–159, 1997.

[Cor92] Thinking Machines Corp. *Connection Machine CM-5 Technical Summary*, Nov. 1992.

[Cor02a] Intel Corporation. *Evolution of Gigabit Technology*, 2002. http://www.intel.com.

[Cor02b] Intel Corporation. *Evolution of Intel Microprocessors: 1971 to 2007*, 2002. http://www.intel.com.

[Cor03a] Jupitermedia Corporation. Memory pricing guide. http://www.sharkyextreme.com/guides/, 2003.

[Cor03b] Standard Performance Evaluation Corporation. http://www.specbench.org, 2003.

[CP97] G. Cabillic and I. Puaut. Stardurst: an environment for parallel programming on networks of heterogeneous workstations. *Journal of Parallel and Distributed Computing*, 40(1):959–969, 1997.

[Den68] P. Denning. The working set model for program behavior. *Communications of the ACM*, 11(5):323–333, 1968.

[Den80] P. Denning. Working sets past and present. *J. IEEE Transactions on Software Engineering*, 6(1):64–84, 1980.

[Des03] Discrete simulation framework (desmo-j). http://www.desmoj.de, 2003.

[Div94] Intel Supercomputer Systems Division. *Paragon User's Guide*, June 1994.

[DO91] F. Douglis and J. Ousterhout. Transparent process migration: Design alternatives and the sprite implementation. *Software-Practice and Experience*, 21(8):757–785, 1991.

[DZ97] X. Du and X. Zhang. Coordinating parallel processes on networks of workstations. *Journal of Parallel and Distributed Computing*, 46(2):125–135, 1997.

[ECGS92] T. Eicken, D. Culler, S. Goldstein, and K. Schauser. Active messages: A mechanism for integrated communication and computation. In *Proceedings of the ISCA'92*, Gold Coast (Australia), 1992.

[(EP04] Escuela Universitaria Politecnica (EPS). *Horarios de los Laboratorios de la EPS de la UdL.* http://alumnes.eup.udl.es/horaris/horaris.html, 2004.

[Fei96] D. Feitelson. Packing Schemes for Gang Scheduling. In *Job Scheduling Strategies for Parallel Processing*, volume 1162 of *Lecture Notes in Computer Science*, pages 89–110. 1996.

[Fei97] D. Feitelson. Job scheduling in multiprogrammed parallel systems. Technical report, IBM Research RC 19970, 1997.

[FFPF03] E. Frachtenberg, D. Feitelson, F. Petrini, and J. Fernández. Flexible coscheduling: Mitigating load imbalance and improving utilization of heterogeneous resources. In *Proceedings of*

the International Parallel and Distributed Processing Symposium (IPDPS'2003), number 17, 2003.

[Fly72] M. Flynn. Some computer organizations and their efectiveness. *IEEE Trans. on Computers*, C(21):115–128, 1972.

[FN97] D. Feitelson and B. Nitzberg. Job characteristics of a production parallel scientific workload on the nasa ames ipsc/860. In *Job Scheduling Strategies for Parallel Processing*, volume 1291 of *Lecture Notes in Computer Science*, pages 215–227. 1997.

[For02] MPI Forum. http://www.mpi-forum.org/docs/docs.html, 2002.

[FPCF01] E. Frachtenberg, F. Petrini, S. Coll, and W. Feng. Gang scheduling with lightweight user-level communication. In *Proceedings of the 2001 International Conference on Parallel Processing (ICPP'2001)*, Valencia, Spain, 2001. Workshop on Scheduling and Resource Management for Cluster Computing.

[FR92] D. Feitelson and L. Rudolph. Gang scheduling performance benefits for fine-grained synchronization. *J. Parallel and Distributed Computing*, 16(4):306–318, 1992.

[FR96] D. Feitelson and L. Rudolph. Toward convergence in job schedulers for parallel supercomputers. In *Job Scheduling Strategies for Parallel Processing*, volume 949 of *Lecture Notes in Computer Science*, pages 1–26. 1996.

[FRS+97] D. Feitelson, L. Rudolph, U. Schweigelshohn, K. Sevcik, and P. Wong. Theory and practice in parallel job scheduling. In *Job Scheduling Strategies for Parallel Processing*, volume 1291 of *Lecture Notes in Computer Science*, pages 1–34. 1997.

[FST94] A. Feldmann, J. Sgall, and S. Teng. Dynamic scheduling on parallel machines. *J. Theoretical Computer Science*, 130(1):49–72, 1994.

[FZ87] D. Ferrari and S. Zhou. An empirical investigation of load indices for load balancing applications. In *12th International Symposium on Computer Performance Modeling, Measurement and Evaluation*, pages 515–528, 1987.

[GBD+94] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM:Parallel Virtual Machine - A User's Guide and Tutorial for Networked Parallel Computing.* MIT Press Pub., 1994.

[GHS+03] F. Giné, M. Hanzich, F. Solsona, P. Hernández, and E. Luque. Multiprogramming level of pvm jobs in a non-dedicated linux now. In *EuroPVM/MPI'2003*, volume 2840 of *Lecture Notes in Computer Science*, pages 577–586. 2003.

[GKP96] G. Geist, J. Kohl, and P. Papadopoulos. Pvm and mpi: a comparison of features. *J. Calculateurs Paralleles*, 8(2):137–150, 1996.

[GRV01] A. Gaito, M. Rak, and U. Villano. Adding dynamic coscheduling support to pvm. In *EuroPVM/MPI'2001*, volume 2131 of *Lecture Notes in Computer Science*, pages 106–113. 2001.

[GSHL01a] F. Giné, F. Solsona, P. Hernández, and E. Luque. Coscheduling under memory constraints in a now environment. In *Job Scheduling Strategies for Parallel Processing (JSSP'01)*, volume 2221 of *Lecture Notes in Computer Science*, pages 41–65. 2001.

[GSHL01b] F. Giné, F. Solsona, P. Hernández, and E. Luque. Memto: A memory monitoring tool for a linux cluster. In *EuroPVM/MPI'2001 Conference*, volume 2131 of *Lecture Notes in Computer Science*, pages 225–232. 2001.

[GSHL02a] F. Giné, F. Solsona, P. Hernández, and E. Luque. Adjusting the lengths of time slices when scheduling pvm jobs with high memory requirements. In *EuroPVM/MPI'2002*, volume 2474 of *Lecture Notes in Computer Science*, pages 156–164. 2002.

[GSHL02b] F. Giné, F. Solsona, P. Hernández, and E. Luque. Adjusting time slices to apply coscheduling techniques in a non-dedicated now. In *Euro-Par'2002 Parallel Processing*, volume 2400 of *Lecture Notes in Computer Science*, pages 234–240. 2002.

[GSHL02c] F. Giné, F. Solsona, P. Hernández, and E. Luque. Minimizing paging tradeoffs applying coscheduling techniques in a linux cluster. In *Vecpar and Parallel Processing Conference*, volume 2565 of *Lecture Notes in Computer Science*, pages 592–607. 2002.

[GSHL03a] F. Giné, F. Solsona, P. Hernández, and E. Luque. Cooperating coscheduling in a non-dedicated cluster. In *Euro-Par'2003 Parallel Processing*, volume 2790 of *Lecture Notes in Computer Science*, pages 212–218. 2003.

[GSHL03b] F. Giné, F. Solsona, P. Hernández, and E. Luque. Dealing with memory constraints in a non-dedicated linux cluster. *The International Journal of High Performance Computing Applications*, 17(1):39–48, 2003.

[GTU91] A. Gupta, A. Tucker, and S. Urushibara. The impact of operating system scheduling policies and synchronization methods on the performance of parallel applications. In *Proceedings of the ACM SIGMETRICS/PERFORMANCE'91*, pages 120–132, 1991.

[Hag88] R. Hagman. Process server: Sharing processing power in a workstation environment. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, pages 104–111, 1988.

[HTI+96] A. Hori, H. Tezuka, Y. Ishikawa, N. Soda, H. Konaka, and M. Maeda. Implementation of gang-scheduling on workstation cluster. In *Job Scheduling Strategies for Parallel Processing*, volume 949 of *Lecture Notes in Computer Science*, pages 126–139. 1996.

[iB03] J. Vime i Bosch. Pvmweb. Trabajo fin de carrera, Dep. Informática, Universitat de Lleida, 2003.

[Int02a] Intel Corporation, http://developer.intel.com/design/pentium/manuals. *IA-32 Intel Architecture Sotware Developer's Manual. Volume 1*, 2002.

[Int02b] Intel Corporation, http://developer.intel.com/design/pentium/manuals. *IA-32 Intel Architecture Sotware Developer's Manual. Volume 2*, 2002.

[Int02c] Intel Corporation, http://developer.intel.com/design/pentium/manuals. *IA-32 Intel Architecture Sotware Developer's Manual. Volume 3*, 2002.

[Jai91] Raj Jain. *The Art of Computer Systems Performance Analisys*. Wiley Pub., 1991.

[Jav] Java. http://java.sun.com.

[KB93] P. Krueger and D. Babbar. Stealth: a liberal approach to distributed scheduling for network of workstations. Technical report, OSU-CISRCI/93-TR6, Ohio State University, 1993.

[KC91] P. Krueger and R. Chawla. The stealth distributed scheduler. In *Proceedings of the International Conference on Distributed Computing Systems*, pages 336–343, 1991.

[Kea94] C. Koelbel and et al. *The High Performance Fortran Handbook.* The MIT Press Pub., 1994.

[Kle76] L. Kleinrock. *Queuing Systems.* Jonh Wiley and Sons Pub., 1976.

[Kwo03] Y-Kwong Kwok. On exploiting heterogeneity for cluster based parallel multithreading using task duplication. *The Journal of Supercomputing*, 25(1):63–82, 2003.

[LAD+96] C. Leiserson, Z. Abuhamdeh, D. Douglas, C. Feynman, M. Ganmukhi, J. Hill, W. Hillis, B. Kuszmaul, M. St Pierre, D. Wells, M. Wong-Chan, S. Yang, and R. Zak. The network architecture of the Connection Machine CM-5. *Journal of Parallel and Distributed Computing*, 33(2):145–158, 1996.

[LG97] R. Lagerstrom and S. Gipp. Psched: Political scheduling on the cray t3e. In *Job Scheduling Strategies for Parallel Processing*, volume 1291 of *Lecture Notes in Compter Science*, pages 117–139. 1997.

[LKK00] W. Leinberger, G. Karypis, and V. Kumar. Memory management techniques for gang scheduling. In *Euro-Par'2000 Parallel Processing*, volume 1900 of *Lecture Notes in Computer Science*, pages 252–262. 2000.

[LLM88] M. Litzkow, M. Livny, and M. Mutka. Condor - a hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, pages 104–111, 1988.

[LS92] M. Litzkow and M. Salomon. Supporting checkpointing and process migration outside of the unix kernel. In *Proceedings of the Usenix Winter Conference*, 1992.

[LS97] S. Leutenegger and Xian-He Sun. Limitations of cycle stealing for parallel processing on a network of homogeneous workstations. *Journal of Parallel and Distributed Computing*, 43(2):169–178, 1997.

[Mat03] The MathWorks Inc., http://www.mathworks.com/products/matlab. *MatLab 6.5*, 2003.

[MB91] J. Mogul and A. Borg. The effect of the context switches on cache performance. In *Proceedings of the fourth International Conference on Arquitectural Support for Programming Languages and Operating Systems*, 1991.

[MCF+98] J. Moreira, W. Chan, L. Fong, H. Franke, and M. Jette. An infraestructure for efficient parallel job execution in terascale computing environments. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, pages 1–14, 1998.

[Mil68] R. Miller. Response time in man-computer conversational transactions. In *Proceedings of the AFIPS Fall Joint Computer Conference*, volume 33, pages 267–277, 1968.

[Mil95] D. Mills. Improved algorithms for syncronizing computer network clocks. *IEEE/ACM Trans. Networks*, 3:245–254, 1995.

[ML91] M. Mutka and M. Livny. The available capacity of a privately owned workstation environment. *J. Performance Evaluation*, 12(4):269–284, 1991.

[MS70] R Meyer and L. Seawright. A virtual machine time -sharing system. *IBM Systems Journal*, 9(3):199–218, 1970.

[MZ95] C. McCann and J. Zahorjan. Scheduling memory constrained jobs on distributed memory parallel computers. In *Proceedings of the ACM SIGMETRICS/PERFORMANCE'95*, pages 208–219, 1995.

[Nea95] J.B. Nanette and et al. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, pages 29–36, Feb 1995.

[Nie00] J. Nielsen. *Usabilidad Diseño de Sitios Web.* Prentice Hall Pub., 2000.

[OCD+88] J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson, and B. Welch. The sprite network operating system. *IEEE Computer*, 21(2):23–26, 1988.

[OSHH96] B. Overeinder, P. Sloot, R. Heederik, and L. Hertzberger. A dynamic load balancing system for parallel cluster computing. *J. Future Generation Computer Systems*, 12(1):101–115, 1996.

[Ous82] J. Ousterhout. Scheduling strategies for concurrent systems. In *Proceedings of 3rd. International Conference of Distributed Computing Systems*, pages 22–30, 1982.

[Pet02] M. Petterson. Linux x86 performance-monitoring counters driver. http://www.csd.uu.se/ mikpe/linux/perfctr, 2002.

[PHFG01] F. Petrini, A. Hoisie, W. Feng, and R. Graham. Performance evaluation of the quadrics interconnection network. In *Workshop on Communication Architecture for Clusters (CAC'01)*, 2001.

[PKB00] R. Poovendran, P. Keleher, and J. Baras. A decision-process analysis of implicit coscheduling. In *Proceedings of the 2000 International Parallel and Distributed Processing Symposium (IPDPS)*, 2000.

[Pol96] A. Polze. How to partition a workstation. In *Proceedings of the IASTED International Conference Parallel and Distributed Computing*, pages 184–187, 1996.

[Pro02] The Berkeley Intelligent RAM Project. http://iram.cs.berkeley.edu, 2002.

[PS96] E. Parsons and K. Sevcik. Coordinated allocation of memory and processors in multiprocessors. In *Proceedings of the ACM SIGMETRICS/PERFORMANCE'96*, pages 57–67, 1996.

[RH98] K. Ryu and J. Hollingsworth. Linger longer: Fine-grain cycle stealing for networks of workstations. In *Proceedings of Supercomputing'98*, 1998.

[Ryu01] K. Ryu. *Exploiting Idle Cycles in Networks of Workstations.* PhD thesis, University of Maryland, 2001.

[Sar] Iostat Sar. http://perso.wanadoo.fr/sebastien.godard.

[SFHL00] F. Solsona, F.Giné, P. Hernández, and E. Luque. Implementing explicit and implicit coscheduling in a pvm environment. In *EurPar'2000*, volume 1900 of *Lecture Notes in Computer Science*, pages 1165–1170. 2000.

[SGHL00] F. Solsona, F. Giné, P. Hernández, and E. Luque. Implementing and analysing an effective explicit coscheduling algorithm on a now. In *Vector and Parallel Processing (VECPAR'2000)*, volume 1981 of *Lecture Notes in Computer Science*, pages 75–88. 2000.

[SGHL01a] F. Solsona, F. Giné, P. Hernández, and E. Luque. Cmc: A coscheduling model for non-dedicated cluster computing. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'01)*, 2001.

[SGHL01b] F. Solsona, F. Giné, P. Hernández, and E. Luque. Coscheduling policies: Simulation vs implementation. In *Proceedings of the 5th. World Multiconference on Systemics, Cybernetics and Informatics (SCI'2001)*, volume XV, pages 579–584, 2001.

[SGHL01c] F. Solsona, F. Giné, P. Hernández, and E. Luque. Predictive coscheduling implementation in a non-dedicated linux cluster. In *Euro-Par'2001*, volume 2150 of *Lecture Notes in Computer Science*, pages 732–741. 2001.

[SGL+00] F. Solsona, F. Giné, J. Lérida, P. Hernández, and E. Luque. Monito: a communication monitoring tool for a pvm-linux environment. In *EuroPVM/MPI'2000 Conference*, volume 1908 of *Lecture Notes in Computer Science*, pages 233–241. 2000.

[Sol02] F. Solsona. *Coscheduling Techniques for Non-dedicated Cluster Computing.* PhD thesis, Dept. Informatica, Universitat Autonoma de Barcelona, 2002.

[SPWC98] P. Sobalvarro, S. Pakin, W. Weihl, and A. Chien. Dynamic coscheduling on workstation clusters. In *Job Scheduling Strategies for Parallel Processing*, volume 1459 of *Lecture Notes in Computer Science*, pages 231–256. 1998.

[SRD01] G. Suh, L. Rudolph, and S. Devadas. Effects of memory performance on parallel job scheduling. In *Job Scheduling Strategies for Parallel Processing*, volume 2221 of *Lecture Notes in Computer Science*, pages 116–132. 2001.

[SS95] K. Schauser and C. Scheiman. Experience with active messages on the meiko cs-2. In *Proceedings of the 9th International Parallel Processing Symposium (IPPS'95)*, pages 140–150, 1995.

[Ste92] M. Steiner. Extending multiprogramming to a dmpp. *J. Future Generation Comput. Syst.*, 8:93–109, 1992.

[SVS02] J. Subhlok, S. Venkataramaiah, and A. Singh. Characterizing nas benchmark performance on shared heterogenous networks. In *11th International Heterogenous Computing Workshop*. IEEE, 2002.

[SW95] P. Sobalvarro and W. Weihl. Demand-based coscheduling of parallel jobs on multiprogrammed multiprocessors. In *Job Scheduling Strategies for Parallel Processing*, volume 949 of *Lecture Notes in Computer Science*, pages 106–126. 1995.

[Tan99] A. Tanenbaum. *Structured Computer Organization*. Prentice Hall International Pub., 1999.

[TLC85] M. Theimer, K. Lantz, and D. Cheriton. Preemptable remote execution facilities for the v-system. In *Proceedings of the 10th ACM Symp. on Operating Systems Principles*, pages 2–12, 1985.

[top02] Top 500 supercomputers sites. http://www.top500.org/, 2002.

[TWY92] J. Turek, J. Wolf, and P. Yu. Approximate algorithms for scheduling parallelizable tasks. In *Proceedings of the 4th Symp. Parallel Algorithms & Architectures*, pages 323–332. June 1992.

[Uni02] University of Auckland, http://www.r-project.org. *The R Project for Statical Computing*, 2002.

[VM02] J. Vetter and F. Mueller. Communication characteristics of large-scale scientific applications for contemporary cluster architecture. In *16th Intl. Parallel & Distributed Processing Symposium*, 2002.

[VS97] Mc. Voy and C. Staelin. *Lmbench: Portable Tools for Performance Analysis*. ftp://ftp.sgi.com/pub/lmbench.tgz, 1997.

[WADC99] C. Wong, A. Arpaci-Dusseau, and D. Culler. Buliding mpi for multi-programming systems using implicit information. In *EuroPVM/MPI'1999*, volume 1697 of *Lecture Notes in Computer Science*, pages 215–222. 1999.

[WMADC99] F. Wong, R. Martin, R. Arpaci-Dusseau, and D. Culler. Architectural requirements and scalability of the nas parallel benchmarks. In *Proceedings of Supercomputing'99*, 1999.

[Wol03] Wolfram Research, http://www.wolfram.com/products-/mathematica/index.html. *Mathematica 5*, 2003.

[WR97] M. Weil and L. Rosen. *Technostress: Coping with Technology @Work @Home @Play.* Jonh Wiley & Sons Pub., 1997.

[YJ01] A. Yoo and M. Jette. A coscheduling technique for large symmetric multiprocessor clusters. In *Job Scheduling Strategies for Parallel Processing*, volume 2221 of *Lecture Notes in Computer Science*, pages 21–40. 2001.

[ZSMF01] Y. Zhang, A. Sivasubramaniam, J. Moreira, and H. Franke. Impact of workload and systems parameters on next generation cluster scheduling mechanism. *IEEE Transactions on Parallel and Distributed Systems*, 12(9):967–985, 2001.