BIBLIOTECA VIRTUAL
MIGUEL DE CERVANTES

Universidad de Valladolid

VSR-COMA: Un Protocolo de Coherencia Cache con Reemplazo para Sistemas Multicomputadores con gestión de memoria de tipo COMA

Diego R. Llanos Ferraris

Tesis de Doctorado

Facultad: Informática

Director: Dr. Benjamín Sahelices Fernández

2000

Universidad de Valladolid
Departamento de Informática

VSR-COMA: Un Protocolo de Coherencia Cache con Reemplazo para Sistemas Multicomputadores con gestión de memoria de tipo COMA

Tesis Doctoral

Diego R. Llanos Ferraris

Abril de 2000

Resumen

La posibilidad de construcción de sistemas multicomputadores débilmente acoplados de memoria compartida distribuida utilizando una red de estaciones de trabajo resulta atractiva por diferentes motivos. En primer lugar, dichos sistemas permiten acelerar la ejecución de aplicaciones paralelas a bajo coste. Otra de sus ventajas es la utilización de un espacio compartido de direcciones entre los diferentes nodos, que posibilita el uso de un paradigma de programación de variables compartidas. Este hecho facilita en gran medida su programación, al evitar que el programador deba encargarse explícitamente de gestionar el intercambio de datos entre los diferentes nodos.

La aplicación a esta clase de sistemas de mecanismos de gestión de memoria de tipo COMA permite mantener la coherencia de los datos presentes en las memorias de cada uno de los nodos, reduciendo el tiempo de acceso a los datos a través de la combinación de mecanismos de replicación y de migración de bloques de datos a los nodos que los utilizan en cada momento. La migración automática de bloques, por otra parte, evita el problema de la distribución inicial de los datos entre los diferentes nodos.

Uno de los principales problemas que presentan los protocolos de tipo COMA es el llamado “problema del reemplazo”. Este problema consiste en el desalojo de un bloque de datos presente en un nodo hacia la memoria de otro nodo, con el objeto de generar espacio libre para admitir un nuevo bloque. Existen diferentes estrategias para seleccionar el nodo destino de una operación de reemplazo. En el presente trabajo se ha realizado un estudio de los mecanismos de reemplazo existentes en los protocolos COMA y se propone una nueva solución, basada en el mantenimiento local en cada controlador de coherencia de la información de estado de las memorias remotas.

La solución que se propone al problema del reemplazo ha sido utilizada para la construcción de un nuevo protocolo de coherencia cache de tipo COMA para su uso en sistemas multicomputador débilmente acoplados. Este protocolo, denominado VSR-COMA, incorpora todas las características necesarias para su utilización en un sistema real: mecanismos de reemplazo de bloques, asociatividad por conjuntos y operaciones atómicas que facilitan la sincronización.

Para estudiar el funcionamiento del protocolo de coherencia se ha desarrollado un simulador que permite estudiar el comportamiento de un sistema VSR-COMA formado por una red de estaciones de trabajo en la ejecución de aplicaciones paralelas. Dicho simulador permite además utilizar diferentes mecanismos de reemplazo, al objeto de comparar su funcionamiento con el del mecanismo propuesto para VSR-COMA.

Los resultados de las simulaciones han permitido establecer que un sistema débilmente acoplado que utilice el protocolo VSR-COMA para el mantenimiento de la coherencia permite acelerar la ejecución de aplicaciones paralelas manteniendo un tráfico reducido en la red de interconexión. Por otra parte, la utilización del simulador nos ha permitido constatar la mejora en el rendimiento obtenida a través del uso de la estrategia de reemplazo de VSR-COMA frente a otras estrategias propuestas hasta la fecha.

Declaración

Dr. D. Benjamín Sahelices Fernández, Profesor Titular de Escuela Universitaria del Departamento de Informática (ATC, CCIA y LSI) de la Universidad de Valladolid,

CERTIFICA

Que D. Diego R. Llanos Ferraris, Ingeniero en Informática por la Universidad de Valladolid, ha realizado bajo mi dirección la Memoria que lleva por título "VSR-COMA: Un Protocolo de Coherencia Cache con Reemplazo para Sistemas Multicomputadores con gestión de memoria de tipo COMA", con el fin de que sea presentada como Tesis para aspirar al grado de Doctor en Informática.

Y para que surta los efectos oportunos, firmo la presente en Valladolid, a 3 de abril del año dos mil.

Agradecimientos

En primer lugar, deseo agradecer profundamente el apoyo que he recibido del Director del presente trabajo, el Dr. Benjamín Sahelices Fernández. En él he encontrado a un excelente investigador y a un gran amigo. También deseo agradecer la ayuda prestada por el Dr. Agustín de Dios Hernández. Sin el interés y el entusiasmo que ambos han puesto en este trabajo difícilmente hubiera podido llegar a buen término.

El área de Arquitectura y Tecnología de Computadores del Departamento de Informática ha puesto todos los medios a su alcance para ayudarme con mi tarea, sobre todo en lo que respecta a la distribución de la carga docente. En este sentido, la contribución de sus miembros en el presente trabajo ha sido decisiva. Quiero aquí dejar constancia de ello, junto con mi profundo agradecimiento.

La ayuda de mis compañeros del Departamento de Informática ha facilitado en gran medida mi trabajo de tesis. El apoyo prestado ha sido muy diverso: desde la utilización de recursos de cálculo pertenecientes a otros grupos de investigación del Departamento hasta los cafés compartidos con el resto de la "Legión Extranjera" en la Politécnica, pasando por las preguntas del tipo "¿qué estás haciendo ahora en lugar de hacer la tesis?" que el Dr. José Manuel Marqués me ha dirigido con una sonrisa tantas veces en estos años. Quedo en deuda con todos ellos.

Deseo agradecer también al Dr. Josep Torrellas, de la Universidad de Illinois en Urbana-Champaign, USA, así como a su equipo de investigación. Los comentarios y sugerencias que he recibido respecto del presente trabajo a lo largo de mi estancia en dicha universidad han resultado muy útiles.

Para llevar a cabo un trabajo de tesis no sólo son necesarios apoyos científicos, sino también humanos. Una tesis supone un esfuerzo personal muy grande, y acaso el precio a pagar a veces sea excesivo. En ese sentido, he tenido la suerte de poder contar con amigos que me han prestado su apoyo en momentos difíciles. Quisiera mencionar aquí a Carlos García, a Javier Ramos, a José y a María del Carmen Poza, a Beatriz y a Marta Huidobro, a Leopoldo Feal y a Paula Álvarez, junto con los ya mencionados Benja y Agustín, agradeciéndoles por haber estado a mi lado en todo momento y especialmente por haberme sacado adelante en mis horas más bajas.

Tengo la inmensa fortuna de tener una familia que me ha apoyado a lo largo de toda mi vida. Mis padres, Teresa Ferraris y Rafael Llanos, se han sacrificado muchos años para que yo finalmente me dedicara a lo que realmente me gusta. Todo se lo debo a ellos, y por eso, como humilde compensación, les dedico este trabajo.

Seguramente no he mencionado a mucha gente que también me ha ayudado con este trabajo. Vaya para ellos mis más sinceras disculpas y mi agradecimiento.

A mis padres

Índice General

1 Introducción **1**

2 Sistemas de memoria compartida distribuida (DSM) **5**
2.1 Introducción 5
2.2 Sistemas de memoria compartida distribuida 7
2.3 Arquitectura DVSM 8
2.4 Arquitectura NUMA 9
2.5 Arquitectura CC-NUMA 10
2.6 Arquitectura COMA 12
2.6.1 Mantenimiento de la coherencia 14
2.6.2 Localización de bloques 16
2.6.3 Reemplazo de bloques 17
2.6.4 Sobrecarga de memoria 18
2.6.5 Otras cuestiones abiertas 18
2.7 Conclusiones 19

3 El reemplazo en los protocolos COMA **21**
3.1 Introducción 21
3.2 El problema del reemplazo 21
3.3 El reemplazo en sistemas COMA existentes 23
3.3.1 Gestión centralizada: el sistema COMA-F 24
3.3.2 Gestión distribuida: el sistema DICE 37
3.3.3 Otros protocolos 42
3.4 Nuestra propuesta: VSR-COMA 45
3.5 Conclusiones 46

4 El protocolo VSR-COMA **47**
4.1 Introducción 47
4.2 Objetivos de diseño 48
4.3 Gestión distribuida de la coherencia 49
4.4 Nomenclatura 52
4.4.1 Protocolo de memoria 52
4.4.2 Protocolo entre nodos 53

4.4.3 Estados de los bloques 54
4.4.4 Nodos 55
4.5 Mantenimiento de la información de reemplazo 55
4.6 Premisas de funcionamiento 57
4.7 Funcionamiento general de VSR-COMA 58
4.7.1 Lectura 58
4.7.2 Escritura 63
4.7.3 Reemplazo 69
4.8 Diagrama de estados 78
4.9 Tratamiento de los eventos entrantes 80
4.9.1 Nomenclatura 81
4.9.2 Solicitud de lectura (evento BusRreq) 82
4.9.3 Respuesta de lectura (evento BusRack) 87
4.9.4 Solicitud de escritura (evento BusWreq) 89
4.9.5 Respuesta de escritura (evento BusWack) 94
4.9.6 Invalidación rápida (evento BusFInv) 97
4.9.7 Solicitud de desalojo (evento BusEXreq) 99
4.9.8 Respuesta positiva de desalojo (evento BusEXack) . 102
4.9.9 Respuesta negativa de desalojo (evento BusEXnak) . 105
4.9.10 Condiciones de carrera (evento BusRACE) 107
4.10 Selección del nodo destino para desalojo 108
4.11 Acceso a datos situados en límites de bloques 111
4.12 Conclusiones 113

5 Simulación de arquitecturas COMA 115
5.1 Introducción 115
5.2 La simulación basada en ejecución 115
5.3 Modelo de programación 117
5.3.1 Estructura de la memoria compartida 117
5.3.2 Las macros Parmacs 118
5.3.3 Estructura de la aplicación paralela 120
5.3.4 Generación del código ejecutable 123
5.4 El simulador EMUCOMA 124
5.4.1 Diseño del simulador 124
5.4.2 Funciones de captura de referencias a memoria compartida 126
5.4.3 Funciones de sincronización 126
5.4.4 Implementación del simulador 129
5.5 Índices de rendimiento utilizados 137
5.6 Análisis de rendimientos empleando el modelo LogP 139
5.7 Conclusiones 141

6 Evaluación comparativa de rendimientos 143
6.1 Introducción . . . 143
6.2 Características de la carga de trabajo . . . 143
6.2.1 FFT . . . 144
6.2.2 LU . . . 145
6.2.3 Radix . . . 145
6.2.4 Ocean . . . 145
6.2.5 Barnes . . . 146
6.2.6 Radiosity . . . 146
6.2.7 Resumen de características de los programas . . . 147
6.3 Rendimiento de VSR-COMA . . . 147
6.3.1 Aceleración . . . 148
6.3.2 Tráfico de red . . . 149
6.4 Comparativa de estrategias de reemplazo . . . 152
6.4.1 Aceleración . . . 154
6.4.2 Tráfico de red . . . 156
6.5 La cuestión de la sobrecarga de memoria . . . 159
6.6 Conclusiones . . . 161

7 Conclusiones 163

A Modelado de VSR-COMA con redes de Petri coloreadas 169
A.1 Introducción . . . 169
A.2 Las redes de Petri . . . 169
A.2.1 Habilitación y disparo de transiciones . . . 170
A.2.2 Las redes de Petri coloreadas . . . 170
A.2.3 Design/CPN . . . 172
A.3 Modelado de VSR-COMA . . . 172
A.3.1 Tipos de datos . . . 173
A.3.2 Estructura del modelo . . . 173
A.3.3 Condiciones iniciales . . . 175
A.4 Jerarquía de páginas . . . 178
A.4.1 Página “Nodo A” . . . 178
A.4.2 Página “Procesador” . . . 179
A.4.3 Página “Controlador” . . . 180
A.4.4 Página “Acceso local en lectura” . . . 183
A.4.5 Página “Acceso local en escritura” . . . 183
A.5 Accesos remotos . . . 183
A.5.1 Página “Acceso remoto en escritura” . . . 186
A.5.2 Página “Selección del bloque a sobreescribir en escritura” . . . 188
A.5.3 Página “Selección del bloque a desalojar” . . . 188
A.5.4 Página “Recepción respuesta de escritura” . . . 191
A.5.5 Página “Acceso remoto en lectura” . . . 192

A.5.6 Página “Recepción respuesta de lectura” 192
A.5.7 Página “Selección del bloque a sobreescribir en lectura” 196
A.6 Reemplazo 196
A.6.1 Página “Comprobación Resultado” 199
A.6.2 Página “Selección del nodo compartido” 199
A.7 Gestión de eventos entrantes 200
A.7.1 Página “BusRreq” 202
A.7.2 Página “BusRack” 202
A.7.3 Página “BusWreq” 206
A.7.4 Página “BusWack” 206
A.7.5 Página “BusFInv” 209
A.7.6 Página “Invalidación de reemplazo” 209
A.7.7 Página “Invalidación de AM local” 209
A.7.8 Página “BusEXreq” 213
A.7.9 Página “BusEXreq Destino” 213
A.7.10 Página “BusEXack” 213
A.7.11 Página “BusEXnak” 213
A.7.12 Página “BusRACE” 213

Bibliografía **221**

Índice de Figuras

2.1 Memoria compartida distribuida. 7
2.2 Sistema de memoria virtual compartida distribuida (DVSM) 9
2.3 Sistema de memoria de acceso no uniforme con caches coherentes (CC-NUMA) 11
2.4 Sistema de acceso a memoria sólo cache (COMA) 13

3.1 Solicitudes de lectura en COMA-F. 26
3.2 Solicitudes de escritura en COMA-F. 28
3.3 Reemplazo en COMA-F. 31
3.4 Borrado de un bloque en COMA-F. 35
3.5 Protocolo de coherencia de DICE 39

4.1 Estructura de un nodo de proceso en VSR-COMA 50
4.2 Situación inicial en una solicitud de bloque para lectura . . . 60
4.3 Recepción de una solicitud de lectura (evento BusRreq) . . . 61
4.4 Recepción de una respuesta de lectura (evento BusRack) . . 62
4.5 Situación inicial en una solicitud de invalidación rápida . . . 64
4.6 Situación final en una solicitud de invalidación rápida 65
4.7 Situación inicial en una solicitud de bloque para escritura . . 67
4.8 Recepción de una solicitud de escritura (evento BusWreq) . 68
4.9 Recepción de una respuesta de escritura (evento BusWack) . 70
4.10 Situación previa al desalojo de un bloque 71
4.11 Recepción de una solicitud de reemplazo (evento BusEXreq) 73
4.12 Recepción de una respuesta de reemplazo (evento BusEXack) 75
4.13 Condición de carrera en una solicitud de reemplazo 76
4.14 Recepción de una respuesta negativa de reemplazo (evento BusEXnak) 77
4.15 Diagrama de estados de VSR-COMA 79
4.16 Detalle del diagrama de estados de VSR-COMA 80

5.1 Fases de tratamiento del fichero fuente de la aplicación. . . . 124
5.2 Diagrama de bloques del sistema a simular por EMUCOMA . 125
5.3 Estructura de procesos del simulador Emucoma 130

6.1 Comparativa de rendimientos de VSR-COMA a diferentes presiones 150
6.2 Evolución del tráfico para FFT, LU y Radix con VSR-COMA 151
6.3 Evolución del tráfico para Ocean, Barnes y Radiosity con VSR-COMA 153
6.4 Comparativa de rendimientos a presiones del 80% 155
6.5 Evolución del tráfico para FFT, LU y Radix a presiones del 80% 157
6.6 Evolución del tráfico para Ocean, Barnes y Radiosity a presiones del 80% 158

A.1 Ejemplo de redes de Petri 171
A.2 Ejemplo de redes de Petri coloreadas 172
A.3 Declaraciones de conjuntos de colores en el modelo 174
A.4 Jerarquía de páginas de VSR-COMA 176
A.5 Página “Sistema” 177
A.6 Página “Nodo A” 179
A.7 Página “Procesador” 181
A.8 Página “Controlador” 182
A.9 Página “Acceso local en lectura” 184
A.10 Página “Acceso local en escritura” 185
A.11 Página “Gestión Remota” 187
A.12 Página “Acceso remoto en escritura” 189
A.13 Página “Selección de bloque a sobreescribir en escritura” . . 190
A.14 Página “Selección de bloque a desalojar” 191
A.15 Página “Recepción de respuesta de escritura” 193
A.16 Página “Acceso remoto en lectura” 194
A.17 Página “Recepción de respuesta de lectura” 195
A.18 Página “Selección de bloque a sobreescribir en lectura” . . . 197
A.19 Página “Reemplazo” 198
A.20 Página “Comprobación Resultado” 199
A.21 Página “SeleccionNodoCompartido” 201
A.22 Página “Gestión de eventos entrantes” 203
A.23 Página “BusRreq” 204
A.24 Página “BusRack” 205
A.25 Página “BusWreq” 207
A.26 Página “BusWack” 208
A.27 Página “BusFInv” 210
A.28 Página “Invalidación de reemplazo” 211
A.29 Página “Invalidación de AM local” 212
A.30 Página “BusEXreq” 214
A.31 Página “BusEXreq Destino” 215
A.32 Página “BusEXack” 216
A.33 Página “BusEXnak” 217
A.34 Página “BusRACE” 219

Índice de Tablas

6.1 Características de los programas Splash-2 utilizados en las simulaciones. 147
6.2 Sobrecarga de memoria de control 160

Capítulo 1

Introducción

En los últimos años se han propuesto diferentes arquitecturas de sistemas multicomputadores basados en la utilización de estaciones de trabajo unidas a través de una red de interconexión. El uso de una arquitectura de memoria compartida distribuida en esta clase de sistemas presenta algunas características que las hacen muy atractivas para la ejecución de aplicaciones concurrentes. En primer lugar, el paradigma de memoria compartida distribuida facilita en gran medida el desarrollo de aplicaciones paralelas. El programador utiliza un modelo de variables compartidas entre todos los nodos del sistema, gestionando el acceso a regiones críticas a través de primitivas de sincronización. Esta arquitectura de memoria es similar a la utilizada en un sistema operativo multitarea, lo que permite la ejecución en un sistema multiprocesador de aplicaciones paralelas originalmente escritas para entornos multitarea sin necesidad de efectuar profundos cambios en el código. Por otra parte, la utilización de componentes comerciales como bloques de construcción de un sistema multicomputador, tanto en lo que respecta a estaciones de trabajo como a la red de interconexión, permite acelerar la ejecución de cargas de trabajo paralelas a un reducido coste.

Una de las principales cuestiones a resolver en esta clase de arquitecturas es la reducción del tiempo de acceso a los datos del espacio compartido de memoria. Dado que los sistemas multicomputador de memoria compartida son sistemas débilmente acoplados, el coste asociado a la transferencia de un bloque de datos de un nodo a otro es muy alto, ya que dicha transferencia se realiza a través de la red de interconexión. Esto hace que deba minimizarse el número de accesos a la red, intentando que los datos estén presentes de antemano en los nodos que vayan a utilizarlos. Ello puede conseguirse con una distribución previa a la ejecución, basada en las características de acceso a los datos de cada aplicación. Para llevar a cabo esta tarea pueden utilizarse tanto elementos software, con la ayuda de compiladores y del sistemas operativo, como hardware, a través de la definición de arquitecturas específicas. Sin embargo, la variabilidad en el patrón de

accesos de las aplicaciones a los datos hacen muy difícil en algunos casos una distribución previa eficiente de los datos entre los diferentes nodos de proceso.

Se ha propuesto una alternativa para minimizar el número de accesos a datos situados en nodos remotos para arquitecturas de memoria compartida distribuida. La arquitectura COMA (*Cache Only Memory Architecture*) permite mantener la coherencia de un sistema de este tipo, buscando además aumentar la probabilidad de que pueda accederse a los datos de forma local. Esto se consigue a través de dos mecanismos básicos. El primero consiste en la utilización de copias de bloques que están siendo accedidos por diferentes nodos para su lectura. Este mecanismo, denominado *replicación*, permite realizar lecturas locales una vez que se ha obtenido una copia del bloque requerido. El segundo mecanismo, denominado *migración*, permite que un nodo que desee modificar un dato del espacio compartido de memoria obtenga la copia principal del bloque en el que se encuentra dicho dato. Por lo tanto, el espacio de direcciones compartido no se distribuye estáticamente entre los nodos al principio de la ejecución, sino que cada nodo va "atrayendo" hacia su módulo de memoria local (denominada *memoria de atracción*) los bloques de datos que utiliza en cada momento, lo que permite resolver la cuestión de la distribución inicial de los datos. La migración se realiza de forma completamente transparente en tiempo de ejecución, por lo que no hace falta conocer con antelación el patrón de accesos de la aplicación a la memoria compartida.

Sin embargo, la utilización de un mecanismo tipo COMA para el mantenimiento de la coherencia en un sistema multicomputador formado por un conjunto de estaciones de trabajo plantea algunos problemas. El primero atañe a la complejidad del protocolo de coherencia, ya que la ubicación de los datos no es fija, lo que obliga a un cuidadoso diseño que permita localizar los datos y asegurar el mantenimiento de la coherencia. Otro problema de gran importancia es el llamado "problema del reemplazo". Cuando un nodo necesita un bloque de datos que no está presente en su memoria de atracción debe solicitarlo al nodo que posea la copia principal de dicho bloque. Antes de realizar la solicitud, el nodo debe disponer de espacio libre en su memoria de atracción para poder almacenar el bloque cuando lo reciba. Si no dispone de espacio libre, el nodo puede verse obligado a desalojar uno de sus bloques hacia otro nodo. La eficiencia de esta operación, denominada "reemplazo", depende entre otros factores de una correcta selección del nodo destino del desalojo, problema para el cual no se dispone aún de una solución definitiva.

El objetivo del presente trabajo es el desarrollo de un protocolo COMA para su uso en un sistema formado por una red de estaciones de trabajo conectadas a través de una red de interconexión de tipo bus, incorporando todas las características requeridas para la utilización en condiciones reales de un sistema multicomputador basado en dicho protocolo. Estas carac-

terísticas son los siguientes: la utilización de un mecanismo de reemplazo de bloques, el uso de memorias de atracción de tipo asociativo por conjuntos y la definición de operaciones atómicas sobre datos presentes en el espacio compartido de direcciones al objeto de facilitar la sincronización de procesos.

El trabajo presentado se estructura en tres partes. En primer lugar, se realiza un estudio en profundidad de las arquitecturas COMA y en particular del problema del reemplazo. La selección del nodo destino de una operación de reemplazo es una cuestión delicada, que influye en la tasa de fallos futura y, por consiguiente, en el rendimiento global del sistema. Se estudian dos soluciones existentes: la selección aleatoria y la selección basada en consulta. Se propone además una solución original al problema del reemplazo, basada en el mantenimiento en cada controlador de coherencia de la información de estado de los bloques situados en las memorias de atracción de los nodos remotos.

En segundo lugar, en el presente trabajo se propone un nuevo protocolo de coherencia, denominado VSR-COMA. Se trata de un protocolo de tipo COMA para sistemas multicomputadores formados por un conjunto de estaciones de trabajo conectadas a través de una red de interconexión de tipo bus. El protocolo VSR-COMA incorpora el nuevo mecanismo de reemplazo propuesto, aunque su diseño modular permite la utilización en su lugar de otros mecanismos de reemplazo. El protocolo VSR-COMA ha sido verificado mediante un modelo construido con redes de Petri coloreadas, lo que ha permitido comprobar su corrección en las etapas iniciales del diseño.

Por último, en el presente trabajo se realiza una simulación de rendimiento en la ejecución de programas paralelos pertenecientes al repertorio Splash-2 sobre un sistema multicomputador formado por un conjunto de estaciones de trabajo conectadas a través de una red de interconexión. Para ello se ha desarrollado un simulador (denominado EMUCOMA) que ha permitido el estudio del comportamiento del protocolo VSR-COMA a diferentes presiones de memoria. Los resultados obtenidos muestran valores de aceleración para 16 procesadores de hasta un factor de 8 para algunos de los programas examinados, utilizando una presión de memoria del 25%, y de hasta 7,6 para presiones del 50%.

Dado que el protocolo VSR-COMA permite la utilización de diferentes mecanismos de reemplazo, se ha utilizado esta característica para comparar el funcionamiento de la estrategia propuesta en este trabajo con el resto de estrategias de reemplazo analizadas, al objeto de comparar su rendimiento. Los resultados indican que la estrategia propuesta para VSR-COMA permite obtener incrementos de la aceleración para algunas cargas de trabajo de hasta un 124% con respecto a la estrategia de selección aleatoria y de hasta un 315% con respecto a la estrategia basada en consulta, utilizando 16 nodos y una presión de memoria del 80%.

La presente memoria se organiza de la siguiente forma. En el capítulo 2 se introduce la clasificación de los sistemas de memoria compartida distribuida, enumerando sus ventajas e inconvenientes, y prestando una especial atención a los sistemas COMA. En el capítulo 3 se estudia en profundidad el problema del reemplazo en los protocolos COMA, describiendo las soluciones propuestas hasta la fecha e introduciendo la nueva solución adoptada en VSR-COMA. El capítulo 4 describe en detalle el protocolo VSR-COMA, tanto en lo que respecta a su funcionamiento general como a la gestión de las diferentes transacciones que lo componen. Se describe además la estrategia de reemplazo que utiliza VSR-COMA. En el capítulo 5 se presenta el simulador EMUCOMA, construido para simular el funcionamiento de diferentes mecanismos de reemplazo a través de la ejecución de aplicaciones en un sistema VSR-COMA. Se describe tanto la estructura del simulador como el modelo de programación de las aplicaciones que lo utilicen. En el capítulo 6 se compara el rendimiento de los mecanismos de reemplazo propuestos en la bibliografía con el utilizado por VSR-COMA en la ejecución de aplicaciones paralelas pertenecientes al conjunto Splash-2. Finalmente, el capítulo 7 enumera las conclusiones.

Capítulo 2

Sistemas de memoria compartida distribuida (DSM)

2.1 Introducción

Las arquitecturas de computación paralela pueden verse como una extensión de las arquitecturas convencionales que permitan la cooperación y la comunicación entre elementos de proceso [20]. Existen dos facetas diferentes en las arquitecturas de computación paralela: por una parte, el *modelo de programación*, que consiste en la imagen conceptual de la máquina que el programador utiliza en la codificación. Por otra parte, la *capa de abstracción de comunicaciones* se encarga de resolver la comunicación y la sincronización de los procesos paralelos a través de un conjunto de primitivas. Estas primitivas pueden estar implementadas a nivel hardware, a nivel del sistema operativo o a nivel de un software de usuario específico para cada máquina que relacione el modelo de programación con las primitivas de comunicación.

Respecto al modelo de programación, existen hoy día tres paradigmas principales [20]:

Modelo de variables compartidas : en este modelo, las comunicaciones entre los diferentes procesadores se realizan a través de accesos a un espacio de direcciones compartido. Entre sus principales ventajas se encuentra la facilidad de programación, ya que la cooperación entre procesos se realiza a través de determinadas variables compartidas, al igual que en un sistema de tiempo compartido.

Modelo de paso de mensajes : en este modelo, la comunicación se realiza a través del envío y recepción de mensajes entre un nodo origen y un nodo destino. Este modelo de programación resulta muy apropiado cuando los bloques de construcción del sistema multiprocesador son sistemas completos, con su microprocesador, su espacio de memoria

privado y sus dispositivos de entrada/salida. Desde el punto de vista del usuario, la comunicación no está integrada en el sistema de memoria como en el caso anterior, sino en el de entrada/salida, ya que debe realizarse de forma explícita.

Paralelización de datos : esta forma de cooperación se basa en un grupo de procesadores que efectúan operaciones sobre diferentes elementos de un conjunto de datos de forma simultánea, intercambiándose la información globalmente antes de continuar. El intercambio de información puede realizarse a través de accesos a memoria compartida o mediante paso de mensajes, ya que esta cuestión no depende del modelo. La aplicación de este modelo está restringido a la resolución de problemas que impliquen el procesamiento en paralelo de datos con una estructura regular, como por ejemplo matrices.

La principal ventaja del modelo de espacio de memoria compartido es que pueden aplicarse los mismos principios de programación que se utilizan en la programación de sistemas de tiempo compartido. Desde el punto de vista del programador, no existe ninguna diferencia entre la sincronización de los diferentes procesos cuando se ejecutan en un sistema multitarea y cuando se ejecutan en un sistema multiprocesador, ya que la sincronización se consigue accediendo a posiciones de memoria de un espacio común de direcciones. En el modelo de paso de mensajes, por el contrario, el programador debe invocar explícitamente las primitivas de sincronización, lo que obliga a reescribir las aplicaciones existentes para sistemas multitarea.

El modelo de programación a utilizar no depende hoy día de la capa de abstracción de comunicaciones que incorpore el sistema: de hecho, es posible construir un modelo de paso de mensajes sobre un sistema de memoria compartida y viceversa [93]. Este hecho ha favorecido una tendencia en el desarrollo de sistemas multiprocesador que consiste en la utilización de las nuevas tecnologías en redes de comunicación para la conexión de grupos de máquinas en *clusters* que operan como una máquina paralela de memoria compartida. Entre las principales ventajas de estos sistemas figura la obtención de índices de rendimiento aceptables en la ejecución de aplicaciones paralelas utilizando computadores de bajo coste [103, 3]. Estos sistemas reciben el nombre de “sistemas multicomputadores débilmente acoplados”. Si se diseña cuidadosamente la topología de interconexión, estos sistemas pueden contener un número de unidades de proceso varios órdenes de magnitud mayor que los sistemas fuertemente acoplados [74].

En los sistemas débilmente acoplados no existen módulos de memoria compartidos entre los diferentes procesadores. Esto obliga a que la comunicación entre los nodos de proceso se realice a través de paso de mensajes. Sin embargo, es posible implementar con ellos un sistema que, basado en el paso de mensajes, simule la existencia de una memoria compartida entre

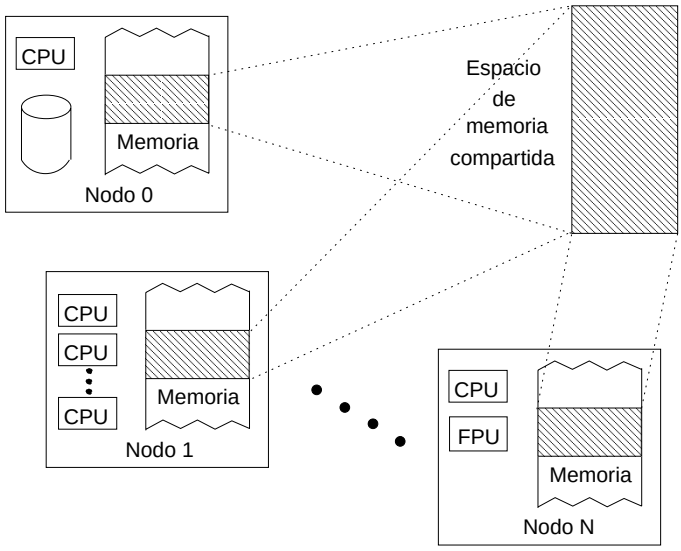


Figura 2.1: Memoria compartida distribuida.

los nodos, lo que permite al programador utilizar técnicas de programación basadas en el paradigma de memoria compartida. Este sistema se denomina "sistema de memoria compartida distribuida".

2.2 Sistemas de memoria compartida distribuida

Los sistemas de memoria compartida distribuida (DSM, *distributed shared memory*) suministran un espacio de direcciones virtual entre los procesadores aunque éstos estén débilmente acoplados (ver figura 2.1). Las principales ventajas de los sistemas DSM son su facilidad de programación, al utilizar el modelo de variables compartidas; la portabilidad de las aplicaciones; el bajo coste asociado a los sistemas débilmente acoplados; y finalmente su mayor escalabilidad respecto a los sistemas fuertemente acoplados, ya que no existen componentes hardware comunes entre los diferentes nodos, sino que la escalabilidad depende del ancho de banda de la red de comunicaciones.

Los sistemas DSM pueden clasificarse en las siguientes arquitecturas:

DVSM (*Distributed Virtual Shared Memory*). En esta clase de sistemas, la capa de software que suministra un espacio compartido de direcciones está soportada a nivel de sistema operativo o de compilador.

NUMA (*Non Uniform Memory Access*). En los sistemas NUMA, cada nodo contiene uno o varios procesadores con caches privadas y un módulo

de memoria que almacena una parte determinada del espacio compartido de direcciones.

CC-NUMA (*Cache-coherent Non Uniform Memory Access*). Los sistemas CC-NUMA buscan reducir el tiempo de acceso a los datos residentes en un nodo remoto utilizando caches que almacenen datos remotos.

COMA (*Cache Only Memory Architecture*). Los sistemas COMA son sistemas en los que el tiempo de acceso a una posición de memoria no depende exclusivamente de su dirección. En los sistemas COMA, los módulos de memoria se comportan como grandes caches que almacenan el conjunto de trabajo utilizado por cada procesador.

Examinaremos brevemente cada una de estas arquitecturas.

2.3 Arquitectura DVSM

Los sistemas DVSM [74] se caracterizan por suministrar un espacio compartido de direcciones mediante mecanismos software. Si el sistema operativo es el encargado de gestionar el espacio compartido, la memoria compartida se implementa como una extensión del sistema de memoria virtual. Al extenderse el mecanismo de paginación utilizado en los sistemas de memoria virtual a un sistema multiprocesador, se mantienen algunas de sus características, entre las que se cuenta el intercambio de páginas como unidad de coherencia.

En el caso de que sea el compilador el encargado de suministrar un espacio compartido de direcciones, éste deberá detectar todas las referencias al espacio común de direcciones y sustituirlas por llamadas a rutinas que se encarguen del mantenimiento de la coherencia entre los datos.

Los sistemas DVSM permiten la migración y réplica de páginas entre nodos mientras se mantiene la coherencia de los datos. Se utiliza la unidad de gestión de memoria (MMU) del procesador para iniciar las acciones indicadas por el protocolo de coherencia (figura 2.2). Estos sistemas presentan dos características destacadas. En primer lugar, los sistemas DVSM utilizan la página como unidad de intercambio entre los nodos, lo que simplifica la gestión de la memoria pero penaliza su rendimiento, debido a problemas de compartición falsa [88]. En segundo lugar, están implementados en software, por lo que pueden utilizarse estrategias de mantenimiento de coherencia más sofisticadas que en otras arquitecturas que utilicen mecanismos hardware para asegurar la coherencia de los datos.

Dos ejemplos destacados de sistemas de memoria virtual compartida distribuida son Ivy y Munin. Ivy [61] es una implementación DVSM sobre un sistema multicomputador débilmente acoplado, construido utilizando una red de estaciones de trabajo. Por su parte, Munin [5, 12] es un sistema

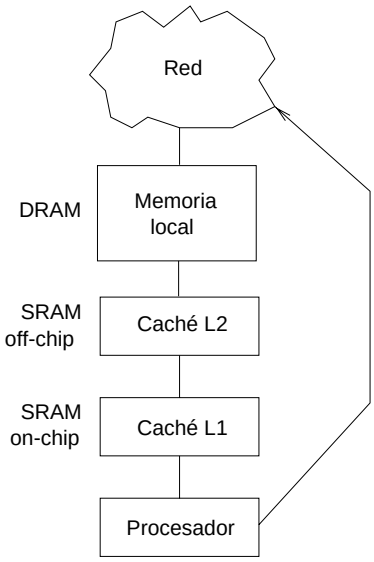


Figura 2.2: Arquitectura de un nodo en un sistema de memoria virtual compartida distribuida (DVSM).

DVSM que incorporó como principal novedad la utilización de diferentes protocolos de coherencia para cada tipo de dato presente en la memoria compartida. La incorporación de modelos de consistencia más relajados [27, 1] permitía reducir el tiempo de latencia en la red. Para conseguir esto, las aplicaciones llevaban anotaciones adicionales que describían el patrón de acceso a dichos datos.

2.4 Arquitectura NUMA

En un sistema multicomputador con memoria compartida físicamente distribuida entre los nodos, el tiempo de acceso a los datos es diferente según si dichos datos se encuentran en la porción de memoria correspondiente al nodo local o en un nodo remoto. En los sistemas NUMA, dicha falta de uniformidad en el acceso se hace visible explícitamente al programador, lo que permite controlar a través de mecanismos software la localización de los datos [9].

En los sistemas NUMA, cada nodo contiene uno o varios procesadores con caches privadas y un módulo de memoria que almacena una parte determinada del espacio compartido de direcciones. Esta parte del espacio de direcciones puede accederse desde cualquier nodo, aunque el tiempo de acceso variará según si el acceso se produce desde el nodo local o desde un nodo remoto. Estos sistemas no necesitan mecanismos para mantener la

coherencia de los datos, ya que sólo existe una copia de cada uno de ellos, y su ubicación es centralizada. Por otra parte, la dirección de memoria especifica el nodo en donde se almacena dicha página. Este nodo recibe el nombre de nodo "hogar" (*home node*). Desarrollos como el llevado a cabo a través del proyecto Platinum [17] demostraron que aplicaciones adecuadamente programadas podían ejecutarse con un rendimiento aceptable.

Pese a la simplicidad de implementación de esta clase de sistemas, su rendimiento se ve seriamente perjudicado debido a la latencia de la red de comunicaciones. Para minimizar el número de accesos remotos, debe prestarse especial atención a la distribución inicial de las páginas de memoria en cada uno de los nodos, una tarea que depende del patrón de acceso a la memoria de la aplicación. Esta tarea debe ser llevada a cabo por el sistema operativo, por el compilador o por el programador, lo que incrementa el coste de desarrollo [8]. También puede mejorarse la localización incorporando elementos hardware, a través del uso de caches consistentes. Esta última posibilidad ha dado lugar a las arquitecturas CC-NUMA.

2.5 Arquitectura CC-NUMA

Los sistemas CC-NUMA (*cache coherent nonuniform memory access*) buscan reducir el tiempo de acceso a los datos residentes en un nodo remoto utilizando en cada nodo una cache de tercer nivel (L3). Esta cache recibe el nombre de "cache remota" (RC) ya que almacena copias de datos remotos que han sido utilizados. El uso de la cache remota permite aumentar el porcentaje de accesos resueltos de forma local [107]. Estos sistemas también se denominan RC-NUMA.

La arquitectura CC-NUMA, al igual que la arquitectura NUMA, utiliza memoria físicamente distribuida entre los nodos. En las máquinas CC-NUMA, cada nodo posee una parte del espacio compartido de direcciones y una memoria cache que puede utilizarse para replicar ítems presentes en cualquier otro nodo. Dado que la memoria física está uniformemente repartida entre los nodos y además su tamaño es igual al de la memoria direccionable, basta con analizar la dirección solicitada para determinar en qué nodo reside el bloque que contiene al dato. Ver la figura 2.3.

Cuando se produce un fallo en la cache local del nodo, el dato se solicita a la memoria local del nodo o a la memoria de otro nodo, de acuerdo con su dirección física. Si el dato en la cache se modifica, se hace necesario actualizar la memoria del nodo en donde resida ese dato. La arquitectura CC-NUMA, por lo tanto, utiliza el bloque como unidad de coherencia. Para cada nodo, el acceso a un bloque viene determinado por el nodo en el que dicho bloque resida, por lo que el acceso es no uniforme.

De lo expuesto se deduce que la cantidad de datos que pueden replicarse localmente (para hacer más eficiente su acceso) se ve limitado por el

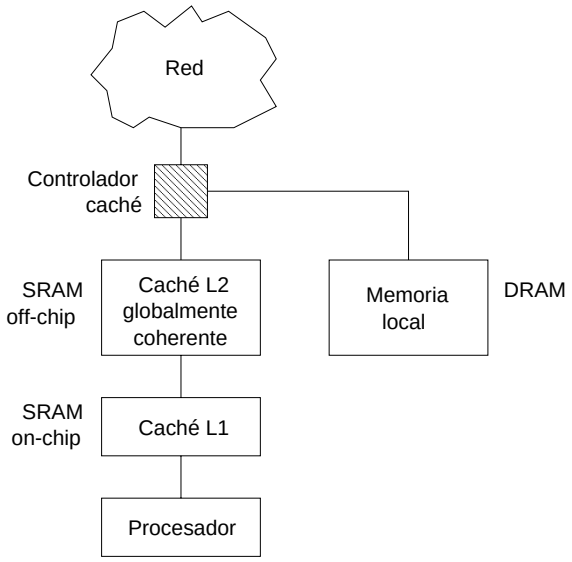


Figura 2.3: Arquitectura de un nodo en un sistema de memoria de acceso no uniforme con caches coherentes (CC-NUMA).

tamaño de la cache (SRAM) de cada nodo, lo que no permite utilizar memoria normal (DRAM) para almacenar copias de los datos [90]. Por otra parte, el rendimiento de las máquinas CC-NUMA depende de forma crítica de la distribución inicial de los datos en las diferentes memorias locales, ya que una distribución incorrecta generaría muchas más solicitudes remotas que locales. Esto también sucede si cada nodo accede a subconjuntos de datos de muy diferente tamaño.

La solución a los problemas derivados de una partición incorrecta de los datos pasa por la *migración de páginas*. Dicho mecanismo, del que se encarga el sistema operativo, permite cambiar la localización física de los datos en función de la ejecución. Sin embargo, este mecanismo es lento, complejo y funciona sólo a granularidad de página, no de bloque [88].

Existen diferentes implementaciones de arquitecturas CC-NUMA. Un ejemplo es el sistema DASH [58, 60, 59], que utiliza una arquitectura a dos niveles: en el nivel superior, cada nodo de proceso está conectado a una red de interconexión de tipo malla. Dentro de cada cluster se utiliza una topología basada en bus para conectar los procesadores. DASH establece una jerarquía de memoria formada por cuatro niveles: la cache de cada procesador, las caches del cluster local, la memoria *home* del bloque y, en el caso de que dicho bloque haya sido modificado por un nodo remoto, el cuarto nivel de la jerarquía lo constituye dicha cache remota.

Otra arquitectura clásica es el sistema MIT Alewife [2]. Esta arquitectura permite la utilización de hasta 512 nodos, a través de una red de inter-

conexión de tipo malla. Entre sus características más destacadas figuran la existencia de mecanismos de reducción de la latencia, como la prebúsqueda, y la posibilidad de utilizar paralelismo de grano fino, lo que posibilita la resolución cooperativa de problemas cuyo conjunto de trabajo sea relativamente pequeño.

El sistema multiprocesador Stanford FLASH [50, 36] utiliza nodos de procesamiento formados por un único procesador, una parte del espacio compartido de memoria y un controlador de coherencia programable implementado en hardware. El sistema STiNG [64], por su parte, es un sistema CC-NUMA disponible comercialmente y basado en el procesador Intel P6. STiNG utiliza nodos de proceso SMP formados por cuatro procesadores, denominados Quads, conectados a través de una interfaz SCI. El sistema STiNG ha permitido obtener buenos resultados en la ejecución de *benchmarks* de procesamiento de transacciones distribuidas (TPC).

En los últimos años se han presentado nuevas técnicas para reducir la latencia de sistemas CC-NUMA, entre las que pueden citarse el uso de mecanismos de encaminamiento adaptativo y de canales virtuales [68], el uso de procesadores de protocolo en lugar de controladores hardware [70], y la utilización de sistemas de migración dinámica de página soportada a nivel de sistema operativo [99].

2.6 Arquitectura COMA

Los sistemas COMA (*Cache Only Memory Architecture*) [35, 22] son sistemas en los cuales el tiempo de acceso a una posición de memoria no depende de su dirección. Para conseguir este objetivo, en las arquitecturas COMA los módulos de memoria se comportan como caches de gran tamaño denominadas "memorias de atracción" (AM) [35].

Como se ha visto en la sección anterior, el rendimiento de los sistemas CC-NUMA se ve perjudicado por el hecho de que los datos situados en nodos remotos no pueden ser almacenados en la memoria local, lo que restringe la cantidad total de datos que pueden replicarse desde los nodos remotos en un instante dado. Por lo tanto, el rendimiento de los sistemas CC-NUMA, al igual que sucede en los sistemas NUMA, se ve perjudicado si se produce una incorrecta distribución inicial de los datos.

En las arquitecturas COMA no existe un dueño fijo para cada bloque de datos, sino que los bloques "migran" hacia los nodos que los requieran. De esta manera, cada nodo atrae hacia su memoria de atracción la parte del espacio de direcciones que utiliza, denominada "conjunto de trabajo" (*working set*).

Al inicio del proceso, cada memoria local contiene una parte del espacio de direcciones compartido de la aplicación. Si a lo largo de la ejecución de la aplicación un nodo solicita un dato que está almacenado en la memoria

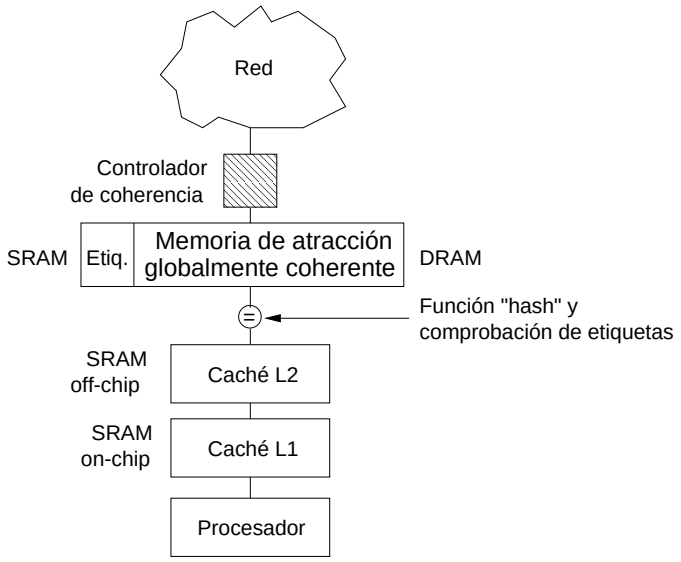


Figura 2.4: Arquitectura de un nodo en un sistema de acceso a memoria sólo cache (COMA).

de atracción de otro nodo, el bloque que lo contiene "migra" a la memoria del nodo que ha hecho la solicitud. A diferencia de la arquitectura CC-NUMA, en la que el almacenamiento de la copia principal de un bloque de datos se realiza en un nodo fijo, la arquitectura COMA permite que cada bloque de datos resida allí donde se lo esté utilizando, al almacenarlo en las memorias de atracción local (figura 2.4).

Este sistema presenta varias ventajas respecto a la arquitectura CC-NUMA. En primer lugar, soluciona uno de los problemas clásicos de CC-NUMA, el relativo a la distribución inicial entre los nodos de los datos que componen el espacio compartido de direcciones. En CC-NUMA, una distribución correcta de los datos es fundamental para reducir la tasa de fallos remota, sobre todo en escritura. En este caso, una escritura provocada por un nodo distinto del propietario del bloque escrito generará siempre un acceso remoto al objeto de actualizar el bloque. Esto hace que la distribución inicial de los datos influya de forma decisiva en el rendimiento. El problema de la distribución inicial es difícil de resolver en las arquitecturas CC-NUMA, ya que dicha distribución depende del patrón de acceso de los nodos a la memoria compartida, y por lo tanto de la aplicación. Por otra parte, puede darse la circunstancia de que el conjunto de bloques al que accede un nodo a lo largo de la ejecución de la aplicación sea mayor que la capacidad de la memoria local, por lo que necesariamente parte del conjunto de trabajo que utiliza ese nodo deberá accederse de forma remota.

La arquitectura COMA soluciona este problema permitiendo que las

copias de cada bloque migren hacia el nodo que la esté utilizando en cada momento. Por lo tanto, la distribución inicial de los datos sólo influirá en el número de accesos que se realicen en los primeros instantes de la ejecución de la aplicación. A medida que los nodos accedan a los bloques que utilizarán, dichos bloques se distribuirán en las memorias de atracción que componen el sistema.

En segundo lugar, la arquitectura COMA soluciona el problema relativo al mantenimiento de réplicas locales de los bloques no utilizados. Muchas veces no es posible realizar una distribución óptima del espacio compartido de direcciones entre las diferentes memorias de un sistema CC-NUMA, debido la mayor parte de las veces a que el patrón de acceso a los datos de la aplicación paralela es irregular. Por lo tanto, puede darse la circunstancia de que muchos de los bloques residentes en la memoria de un nodo CC-NUMA nunca sean utilizados por dicho nodo, ocupando un espacio que no se aprovecha de forma local. La arquitectura COMA permite que un bloque pueda ser desalojado de la memoria de atracción si ese espacio es necesario para almacenar otro bloque, lo que contribuye a incrementar la tasa de acierto local.

Un tercer problema de las arquitecturas CC-NUMA es el reducido tamaño máximo de las caches remotas. En COMA, toda la memoria de atracción se comporta como una memoria cache. Al ser memoria local, es posible incluso implementar mecanismos software de paginación a disco, aunque convenga evitarlos por la sobrecarga en el tiempo de ejecución.

Sin embargo, las arquitecturas COMA no están exentas de algunas desventajas. Entre las principales podemos citar el incremento de la complejidad de los protocolos de coherencia con respecto a los utilizados en CC-NUMA; la necesidad de un sistema eficiente de localización y reemplazo de bloques; o la necesidad de almacenar información de control asociada a los bloques presentes en cada memoria de atracción, lo que conduce a una cierta sobrecarga de la memoria de control. Examinaremos estas cuestiones en las secciones siguientes.

2.6.1 Mantenimiento de la coherencia

El encargado del mantenimiento de la coherencia en los sistemas COMA es un módulo denominado “controlador de coherencia” (ver figura 2.4). Dicho controlador se encarga de mantener la coherencia de los datos presentes en la memoria de atracción, así como de resolver los accesos a nodos que no están presentes en la AM local.

Para facilitar la replicación de los datos en los sistemas COMA se define para cada bloque una única copia “principal” o “maestra”, de la cual se obtienen las copias utilizadas por los restantes nodos. Por otra parte, el nodo que posee en su memoria de atracción la copia maestra de un bloque es el nodo *propietario* de dicho bloque.

Si un nodo solicita para lectura un bloque que no se encuentra en la AM local, el controlador de coherencia se encarga de localizar la copia maestra de dicho bloque en el sistema y solicitar una copia. De esta manera, el bloque es *replicado* en la memoria de atracción local. Si el nodo local debe realizar nuevos accesos en lectura a ese bloque, dichos accesos podrán resolverse de forma local [53]. Por otra parte, existen técnicas hardware que permiten reducir la latencia asociada a las lecturas remotas [105].

Si el nodo local solicita para escritura un bloque del cual no es propietario, el controlador de coherencia se encargará de localizar la copia principal del bloque y solicitarla para escritura. En este caso, el controlador de coherencia del nodo propietario del bloque no sólo envía una copia del bloque, sino que invalida su propia copia. De esta manera, el nodo que solicitó el bloque en escritura pasa a ser el nuevo propietario del bloque. Según la política de actualización del resto de copias que utilice el protocolo, el nuevo nodo propietario puede invalidar las restantes copias o bien enviar una versión actualizada del dato a los nodos que posean copias sin actualizar. Existen arquitecturas en las que es posible elegir la política de actualización a utilizar [82].

Además de satisfacer las operaciones de lectura y escritura, el controlador de coherencia del nodo local se encargará de administrar el espacio disponible en la memoria de atracción. Un problema surge cuando la memoria de atracción está llena y se requiere espacio adicional para almacenar un bloque solicitado por el nodo local. En este caso pueden darse dos circunstancias: que en la AM local existan copias de bloques cuyos propietarios sean remotos, o que el nodo local sea propietario de todos los bloques presentes en su AM. La primera es la más sencilla de resolver, ya que en este caso el controlador de coherencia podrá sobreescribir alguno de dichos bloques, almacenando el nuevo bloque.

Sin embargo, si el nodo local es propietario de todos los bloques presentes en la AM local, no pueden sobreescribirse ninguno de dichos bloques, ya que es posible que sean las últimas copias válidas. En este caso será necesario *desalojar* un bloque hacia otro nodo. Esta situación es diferente de la migración provocada por una falta remota en escritura, ya que el nodo destino del desalojo no tiene interés en recibir ese bloque. El problema es doble: el controlador de coherencia debe seleccionar tanto el bloque a desalojar de entre todos los presentes en la AM como el nodo destino del desalojo.

A la operación consistente en desalojar un bloque al objeto de recibir otro bloque se la denomina *reemplazo*. Como veremos, el problema del reemplazo no ha sido aún resuelto satisfactoriamente, ya que los mecanismos de selección del nodo destino utilizados distan de ser óptimos. La operación de reemplazo supone una sobrecarga de trabajo para el sistema que debe ser reducida en lo posible. La sobrecarga debida al reemplazo puede reducirse aumentando el tamaño de las memorias de atracción, con

lo que aumentará el espacio libre destinado a la replicación. Otra solución es utilizar un mayor número de vías en la asociatividad por conjuntos de las memorias de atracción [46].

A continuación examinaremos brevemente algunas de las principales características que diferencian a las arquitecturas COMA de otros sistemas de memoria compartida distribuida.

2.6.2 Localización de bloques

Dado que las memorias de atracción se comportan como memorias cache, la dirección de un bloque no indica su localización, sino que sólo sirve como un identificador del bloque [22]. La memoria de atracción contiene etiquetas que almacenan el identificador y el estado del bloque asociado a cada uno de los marcos de bloque que componen la memoria. Dado que el identificador no indica la localización del bloque, es necesario disponer de algún mecanismo que permita a cada controlador de coherencia localizar un bloque determinado. El mecanismo de localización está ligado a la topología de la red de interconexión. Los primeros sistemas COMA propuestos utilizaban una topología en árbol, con los nodos de proceso ocupando las hojas del mismo. Esta organización de tipo jerárquico, bien sea de anillos como en KSR-1 [69, 25, 26] o de buses como se propuso en DDM [35], utiliza un sistema de directorios por niveles. Un *directorio* consiste en una estructura de datos que almacena en qué nodos se encuentra cada uno de los bloques de memoria [13, 91]. Cada nivel de la jerarquía incluye un directorio con información acerca del estado de los bloques presentes en todos los nodos que dependen de ese nivel. Para encontrar un bloque remoto, cada nodo deberá enviar una solicitud que va subiendo en la jerarquía hasta que en un nivel determinado el directorio identifique el bloque como perteneciente a un nodo situado por debajo en su jerarquía.

La utilización de topologías jerárquicas permite reducir el número de nodos conectados a un mismo bus, lo que aumenta la escalabilidad del sistema. A cambio, las topologías jerárquicas presentan unos tiempos de latencia que no son fijos, ya que la solicitudes de bloques pueden potencialmente recorrer toda la jerarquía de nodos.

En otros diseños más recientes de sistemas con gestión de memoria de tipo COMA, como COMA-F [45], DDMlite [71], DICE [15] y COMA-BC [87, 84], se utilizan enfoques no jerárquicos, a través de topologías basadas en bus [55]. Todos los nodos se conectan al mismo nivel, lo que además de homogeneizar el acceso a bloques remotos permite utilizar diferentes mecanismos para la localización de los bloques. Estos tres sistemas plantean tres alternativas diferentes. COMA-F propone la utilización de directorios en posiciones fijas. Se define para cada bloque un nodo *home*, que contendrá la información de directorio relativa a ese bloque. Aunque los bloques pueden migrar, las entradas de directorio permanecen en los nodos origi-

nales. Por lo tanto, un nodo que desee obtener un bloque deberá dirigirse al nodo *home* de dicho bloque para que reenvíe la solicitud al nodo correspondiente.

El sistema DICE aprovecha la capacidad de difusión de la topología basada en bus para establecer un mecanismo de tipo *snoopy*, en donde todos los controladores de coherencia asociados a los nodos reciben todos los mensajes que se intercambian en el sistema. Cada bloque tiene un único propietario, no fijo. Cuando un nodo desee solicitar un bloque remoto deberá colocar su solicitud en el bus y esperar a que el nodo propietario del bloque responda. Para su correcto funcionamiento, DICE requiere que el bus pueda ser "reservado" por el nodo solicitante hasta que la respuesta haya llegado, para así evitar condiciones de carrera. En consecuencia, no se permite el solapamiento de solicitudes.

COMA-BC, por su parte, utiliza un enfoque híbrido *snoopy*-directorios. Cada nodo de proceso mantiene un directorio con la localización de todos los bloques del sistema. Por otra parte, el protocolo aprovecha la capacidad de difusión del bus para enviar mensajes que sean recibidos por todos los nodos, reduciendo así el ancho de banda necesario para el mantenimiento de la coherencia. Debido a los mínimos requisitos que COMA-BC impone al funcionamiento del bus común, esta arquitectura permite la construcción de sistemas basados en una red de estaciones de trabajo utilizando gestión de memoria de tipo COMA.

2.6.3 Reemplazo de bloques

Dado que las memorias de atracción funcionan como memorias cache, cuando se pretende introducir un nuevo bloque y el conjunto correspondiente de la memoria de atracción local está lleno, debe desalojarse un bloque. Esta operación de desalojo de un bloque para introducir otro se denomina *reemplazo*. En las memorias cache se garantiza que la jerarquía de memoria inferior tendrá espacio suficiente para almacenar el bloque desalojado de la cache. El problema en los sistemas COMA es más complejo, ya que no existe un nivel de memoria inferior al que las memorias de atracción puedan recurrir. En su lugar, el controlador de coherencia deberá seleccionar una memoria de atracción remota y enviarle el bloque.

Cuando un nodo desee efectuar una operación de reemplazo y el bloque a desalojar (llamado "bloque víctima") haya sido modificado, el nodo tiene la seguridad de que ese bloque no puede descartarse. Sin embargo, a diferencia de lo que ocurre en las memorias cache, un bloque no modificado no puede sobreescribirse sin más, ya que puede suceder que dicho bloque sea la última copia presente en el sistema. Para garantizar que nunca se sobreescriba la última copia de un bloque, se etiqueta una de ellas como la "copia maestra". Si el bloque a descartar no es la copia maestra, el nodo tiene la seguridad de que existen otras copias en el sistema. De no ser

así, deberá ponerse en marcha el mecanismo de reemplazo.

En general, los algoritmos de reemplazo complican enormemente el funcionamiento de los protocolos de coherencia en los sistemas COMA [21]. Examinaremos con detalle este problema en el capítulo siguiente.

2.6.4 Sobrecarga de memoria

En las arquitecturas COMA, la suma de los tamaños de todas las memorias de atracción del sistema es mayor que el tamaño del espacio compartido de direcciones. Esto permite la replicación de bloques para su lectura, lo que reduce la latencia en el acceso a bloques remotos. El cociente entre el tamaño del espacio compartido de direcciones y el tamaño de la suma de las memorias de atracción se denomina *presión de memoria* [51]. Dicha presión de memoria se expresa en términos porcentuales. Si la presión de memoria es del 60 %, significa que el 40 % del tamaño total de las memorias de atracción está disponible para la replicación de bloques. Tanto el tráfico debido al desalojo de bloques como el número de accesos remotos aumentan con la presión de memoria [46].

Al aumento del tamaño de las memorias de atracción para permitir la replicación de los bloques se le añade la necesidad de mantener en cada nodo la información de estado y de etiqueta correspondiente a los bloques presentes en la memoria de atracción, así como la información que sea necesaria para satisfacer las solicitudes que deban resolverse de forma remota. La información que es necesario almacenar depende fundamentalmente del mecanismo de localización de bloques que cada sistema utilice, así como del mecanismo de reemplazo.

2.6.5 Otras cuestiones abiertas

Como se ha visto en la sección 2.6, los sistemas de memoria compartida de tipo COMA presentan algunas ventajas respecto a otras organizaciones de memoria compartida distribuida. Sin embargo, nada se ha dicho respecto a la implementación real de los controladores de coherencia y de las memorias de atracción. Para ello se han establecido diferentes enfoques, de los que destacan dos corrientes principales, denominadas “COMA hardware” y “COMA software” [88]. La primera está formada por sistemas fuertemente acoplados, en donde el control de la coherencia se realiza a través de módulos hardware diseñados especialmente. Estos sistemas utilizan memorias de atracción implementadas en SRAM, al objeto de mejorar los tiempos de respuesta. Podemos encontrar un ejemplo de esta clase de sistemas en DICE [15]. Por otra parte, los sistemas COMA software, como Simple COMA [90, 89] o Multiplexed Simple COMA [4], buscan trasladar parte de la complejidad de los mecanismos de reemplazo a módulos software, aunque mantienen el control de la coherencia en módulos hardware

por razones de eficiencia. Estos sistemas suelen utilizar la página como unidades de coherencia, existiendo estudios comparativos de sus ventajas e inconvenientes [14].

Otra de las cuestiones abiertas en COMA -que no abordaremos en el presente trabajo- es la de la utilización de clusters de procesadores conectados a cada memoria de atracción, en lugar de nodos de proceso formados por un único procesador. Este es el enfoque utilizado en el proyecto I-ACOMA (Illinois Aggresive COMA) [97, 98, 104].

Sin embargo, podemos considerar que el paradigma de funcionamiento de los sistemas COMA no está restringido al uso de módulos hardware para el control de la coherencia. Cabe imaginar un sistema formado por un conjunto de estaciones de trabajo unidos por una red de interconexión que permita la utilización de un espacio compartido de direcciones distribuido con un mecanismo de gestión de tipo COMA. Al no disponerse de un control hardware sobre el medio de interconexión, el protocolo COMA se complica enormemente, ya que tiene que hacer frente a condiciones de carrera que no aparecen en los protocolos utilizados en sistemas fuertemente acoplados. COMA-BC [84] demostró la viabilidad de un sistema de este tipo basado en bus común, que presenta como principal ventaja la ausencia de restricciones en su implementación, posibilitando que el control de la coherencia se realice tanto por módulos hardware como software, e inclusive de forma híbrida, utilizando para ello estaciones de trabajo y sistemas de interconexión disponibles comercialmente a un reducido coste. La eficiencia de un sistema de este tipo en la ejecución de cargas de trabajo en condiciones reales es una cuestión abierta, a la que el presente trabajo pretende aportar algunas respuestas. Para ello se definirá un nuevo protocolo de coherencia, denominado VSR-COMA, con todas las funcionalidades necesarias para su utilización en un sistema formado por estaciones de trabajo unidas por una red de bus común.

2.7 Conclusiones

En el presente capítulo se ha hecho una revisión general de los sistemas de memoria compartida distribuida, señalando sus ventajas e inconvenientes y describiendo algunas de las principales implementaciones existentes. De entre las arquitecturas vistas, hemos centrado nuestra atención en la arquitectura COMA, al poseer algunas características que la hacen muy adecuada para la ejecución de aplicaciones paralelas con patrones de acceso complejos a los datos.

De entre las principales cuestiones abiertas en los sistemas COMA, destaca el problema del reemplazo. El desalojo de un bloque a un nodo remoto es una tarea que introduce importantes latencias en los accesos a datos remotos. En el siguiente capítulo analizaremos el problema del reemplazo en

detalle, describiendo las soluciones planteadas hasta la fecha y presentando VSR-COMA, un protocolo COMA con una gestión elaborada del reemplazo que permite reducir notablemente la latencia debida a dicha operación.

Capítulo 3

El reemplazo en los protocolos COMA

3.1 Introducción

Tras haber examinado en el capítulo anterior la clasificación de las arquitecturas de memoria compartida distribuida y haber descrito los conceptos básicos que caracterizan a la arquitectura COMA, vamos a centrar nuestra atención en uno de los principales problemas que afecta a dichos protocolos: el problema del reemplazo.

De entre todas las cuestiones abiertas relacionadas con las arquitecturas COMA, la relativa al reemplazo es una de las más destacadas. Dado que en las arquitecturas COMA no existe un nivel de memoria al que pueda desalojarse un bloque cuando se requiera espacio, el desalojo deberá hacerse hacia la memoria de atracción de otro nodo. Esto supone tomar básicamente dos decisiones: qué bloque desalojar y hacia qué nodo desalojarlo. De lo acertado de ambas decisiones depende la reducción de faltas en el futuro y, por extensión, la mejora en la velocidad de ejecución de las aplicaciones.

En este capítulo examinaremos con detalle el problema del reemplazo y las diferentes soluciones propuestas en los protocolos COMA existentes. Finalmente, introduciremos una solución nueva, la que propone el protocolo VSR-COMA, y que pretende eliminar algunas de las limitaciones de los mecanismos de reemplazo presentados hasta la fecha.

3.2 El problema del reemplazo

Como hemos dicho en la sección 2.6.3, el problema del reemplazo surge cuando es necesario almacenar un bloque en un conjunto determinado de la memoria de atracción y no hay espacio disponible para ello, ya que todos los marcos de bloque de dicho conjunto se encuentran ocupados por

bloques de los que el nodo es el propietario. En este caso el problema a resolver es doble: qué bloque desalojar y hacia qué nodo desalojarlo.

La primera cuestión admite una solución sencilla. Dado que el objetivo del reemplazo es desalojar un bloque para poder almacenar otro, el bloque a desalojar deberá pertenecer al mismo conjunto de la memoria de atracción que el bloque que se desea pedir. En las memorias de atracción con correspondencia directa, esto significa desalojar obligatoriamente el bloque que está ocupando el marco requerido. En el caso de las memorias de atracción asociativas por conjuntos, el problema queda reducido al seleccionar uno de los bloques del conjunto correspondiente. Finalmente, en las memorias de atracción completamente asociativas la selección del bloque a desalojar deberá hacerse teniendo en cuenta todos los marcos de bloque.

La selección del marco de bloque que se utilizará para almacenar el nuevo bloque en el conjunto correspondiente puede hacerse exclusivamente en función de la información de estado de dicho conjunto [72]. En general, el criterio de selección adoptado en los protocolos COMA con reemplazo utiliza las siguientes pautas:

1. Si existe un marco de "inválido" (es decir, un marco de bloque que no almacena un bloque válido), se selecciona para que almacene el nuevo bloque.

2. Si no es posible, se selecciona un marco de bloque que almacene un bloque del que existen copias en otros nodos.

3. Si no es posible, se selecciona un marco de bloque que almacene un bloque del que no existen copias en otros nodos.

El primer caso es trivial: si existe un marco de bloque cuyo estado es "inválido", existe espacio para almacenar el bloque que va a solicitarse, por lo que no se realiza una operación de desalojo propiamente dicha. El tercero de los casos obliga siempre a desalojar el bloque: corresponde a una situación en la que el nodo posee la única copia válida del bloque, por ejemplo debido a que ha modificado su contenido. Dicha copia no puede descartarse, debiendo ser enviada a otro nodo.

El segundo caso es el que admite mayor cantidad de matices. En general, si existen varias copias de un bloque parecería que la copia puede descartarse sin más. Sin embargo, hay una cuestión importante a tener en cuenta. Dado que el estado de dicho bloque será "compartido", indicando que existen varias copias, puede darse el caso de que todos los nodos que poseen una copia decidan simultáneamente sobreescribirla, basándose en el hecho de que existen otros nodos con una copia válida. Como veremos, este problema se soluciona en la práctica designando una de las copias como la "copia maestra": si un nodo posee la copia maestra de un bloque, deberá transferir su propiedad a otro nodo. Si la copia a desalojar no es

la copia maestra, el nodo puede sobreescribirla sin más, ya que se tiene la seguridad de que no serán eliminadas todas las copias del bloque en el sistema.

La segunda cuestión, la relativa a la selección del nodo destino de una operación de desalojo, dista mucho de tener una solución única. Esto es así debido a que, mientras que la selección del bloque a desalojar puede ser realizada localmente, al disponer el nodo de toda la información necesaria, una selección óptima del nodo destino sólo puede hacerse conociendo el estado en el que se encuentran todas las memorias de atracción del sistema, así como el patrón de accesos que cada uno de los nodos seguirá para acceder a la memoria compartida. Si el nodo que desea desalojar el bloque conociera toda esta información, su selección del nodo destino permitiría reducir al máximo el número de faltas remotas en el futuro, mejorándose el rendimiento del sistema.

Sin embargo, la información que posee cada uno de los nodos del sistema es necesariamente incompleta, ya que no conoce la evolución de las memorias de atracción remotas ni el patrón de accesos de la aplicación. En general, la cantidad de información que cada nodo tiene a su disposición depende del mecanismo de localización de bloques implementado. Si la localización se realiza mediante directorios centralizados, la información relativa a la propiedad de cada bloque se encuentra reunida en un nodo a través del cual pasan todas las solicitudes remotas. Si la localización se realiza a través de mecanismos de tipo *snoopy*, no existe un nodo que centralice la toma de decisiones, por lo que el nodo que desee desalojar el bloque deberá arbitrar algún mecanismo que permita conocer el destino más apropiado para el bloque.

En lo sucesivo nos centraremos en el estudio del reemplazo en protocolos basados en bus común. En la siguiente sección veremos en detalle un ejemplo de cada uno de los mecanismos de localización descritos: un sistema de gestión centralizada basado en directorios (COMA-F) y un sistema de gestión distribuida basado en *snoopy* (DICE). Para cada uno de ellos describiremos el funcionamiento del protocolo y examinaremos las soluciones que se proponen al problema del reemplazo. Haremos también algunas referencias a otros protocolos COMA propuestos para su utilización en bus común. Finalmente, en la sección 3.4 introduciremos el protocolo VSR-COMA, un protocolo para bus común que presenta una nueva solución al problema del reemplazo y cuya descripción detallada se realizará en el siguiente capítulo.

3.3 El reemplazo en sistemas COMA existentes

En esta sección examinaremos el funcionamiento de algunas arquitecturas COMA en bus común y las soluciones que se han propuesto para el pro-

blema del reemplazo. Hemos escogido para ello dos arquitecturas representativas: el sistema COMA-F, con un sistema de localización de bloques basado en directorios, y el sistema DICE, con mecanismo *snoopy*. Tras la descripción de ambos sistemas, comentaremos brevemente algunos otros sistemas COMA propuestos.

3.3.1 Gestión centralizada: el sistema COMA-F

El protocolo COMA-F ha sido desarrollado en la Universidad de Stanford [94, 31, 45]. Se trata de un protocolo para un sistema COMA en bus común no jerárquico. El objetivo es reducir la sobrecarga de memoria resultante de la replicación de los datos. Para ello utiliza una cache entre la memoria de atracción y el procesador que no cumple la llamada *propiedad de subconjunto*, esto es, la cache puede contener datos que no se encuentren en la memoria de atracción del procesador. De esta manera se permite que existan datos en la cache del procesador sin que existan en la memoria de atracción local. Relajando la propiedad de subconjunto se consigue replicar datos en la cache del procesador reduciendo la sobrecarga de replicación de la memoria de atracción.

La utilización de una cache de procesador de esta clase obliga al establecimiento de dos protocolos: uno para mantener la coherencia de los bloques entre las diferentes memorias de atracción (denominado *internode protocol*) y otro que mantenga la coherencia entre la cache del procesador y la memoria de atracción (*intranode protocol*). En lo sucesivo sólo nos ocuparemos del primero de ellos, ya que el segundo sólo se encarga de asegurar la validez de los datos presentes en el mencionado nivel de cache intermedio.

Descripción general

El protocolo utilizado en la arquitectura COMA-F está basado en invalidaciones y utiliza directorios centralizados para mantener la información de estado de los bloques. La arquitectura COMA-F es no jerárquica, lo que, unido a la utilización de directorios centralizados, permite utilizar cualquier topología de conexión entre los nodos, no necesariamente la de bus común.

Cada bloque tiene un nodo *home*, que es el que contiene inicialmente al bloque. El nodo *home* será el encargado de mantener la información de directorio correspondiente a ese bloque, indicando el número de copias del bloque que existen en el sistema y su localización. En todas las transacciones que se realicen sobre un bloque interviene el nodo *home*.

El protocolo admite reemplazo y el uso de primitivas de sincronización. Además, se establece un conjunto de transacciones para la lectura y escritura DMA. Dichas operaciones se realizan de forma directa, sin intervención

por parte del procesador. Por lo tanto, deben establecerse un conjunto de transacciones del protocolo que permitan la lectura y escritura DMA sin que la coherencia de los datos se vea afectada.

El protocolo COMA-F admite además la posibilidad de "borrar" bloques pertenecientes al espacio de direcciones compartido de la aplicación, por ejemplo debido a una transferencia a un dispositivo a través de una operación DMA. También puede darse el caso contrario, es decir, la "inyección" de un bloque del que no existía una copia válida en ninguna de las memorias de atracción del sistema, pese a pertenecer dicho bloque al espacio de direcciones compartido de la aplicación.

Estados y transacciones

Cada bloque de la memoria cache puede estar en uno de cuatro estados posibles:

- *Invalid*: El bloque no es válido.
- *Shared*: El bloque puede leerse y se comparte con uno o más nodos.
- *Master Shared*: El bloque puede leerse y se comparte con cero o más nodos.
- *Exclusive*: El bloque puede leerse y escribirse. No se comparte con ningún otro nodo.

El nodo que contenga un bloque cuyo estado sea *Shared* se considera un nodo que posee una copia "compartida" de ese bloque. Si el estado es *Exclusive* o *Master Shared*, se considera que esa copia es la "copia maestra" de ese bloque.

La memoria de directorio almacena el estado de los bloques asignados a ella. Dicho estado incluye:

- El **estado** del bloque: puede ser inválido, compartido o exclusivo.
- Una **lista de compartición** que contiene a los nodos que poseen copias del bloque.
- Una **identificación del maestro**, que permite saber cuál de los nodos contiene la copia maestra del bloque.

Cabe destacar aquí que, mientras que las copias de los bloques migran a los nodos que la necesiten, la información de directorio permanece en el nodo *home* a lo largo de la ejecución de la aplicación.

La coherencia se mantiene estableciendo la restricción de que sólo pueda procesarse una única *transacción* (formada por los mensajes de solicitud

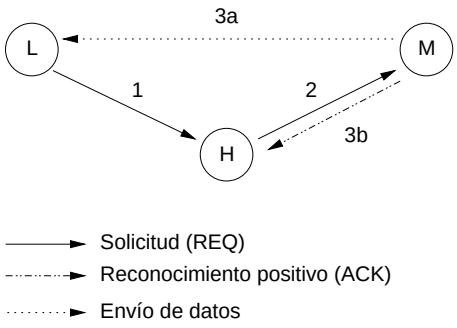


Figura 3.1: Solicitudes de lectura en COMA-F, según [45].

y respuesta) simultáneamente para un bloque dado. Por lo tanto, cada transacción sobre un bloque debe completarse antes de que se pueda iniciar la siguiente sobre el mismo bloque.

En los apartados siguientes veremos cuáles son las transacciones que se utilizan en el protocolo COMA-F para compartir los datos entre las diferentes memorias de atracción. Dichas transacciones se agrupan en tres categorías básicas: solicitud de bloque para lectura, solicitud de bloque en modo exclusivo (es decir, solicitud para escritura) y reemplazo de bloques. Además de estos tres tipos de transacciones, el protocolo define dos tipos adicionales necesarios para la gestión de mecanismos DMA: lectura / escritura DMA y borrado de bloque.

Protocolo de coherencia

En las siguientes figuras, designaremos como L al nodo que realiza la solicitud; H será el nodo que contenga la entrada de directorio correspondiente a ese bloque (nodo *home*), que como hemos visto no tiene por qué coincidir con el nodo maestro; M será el nodo que contenga la copia maestra (*master shared*) del bloque; y S será el nodo que contenga una copia *shared* del bloque.

Solicitudes de lectura Cuando se produce un fallo en lectura en la AM local, el nodo local solicita al nodo que contiene la entrada de directorio correspondiente una copia del bloque. Pueden darse dos casos: que el estado de dicho bloque en el nodo propietario sea *master shared* o que sea *exclusive*. En ambos casos las transacciones a realizar son las mismas: la diferencia está en la actualización de la información de directorio. En una solicitud de lectura las transacciones a realizar son las siguientes (figura 3.1):

1. El nodo local envía una solicitud de lectura al nodo *home*.

2. El nodo *home* reenvía la solicitud al nodo que contiene la copia (maestra o exclusiva) del bloque. Además, añade el nodo local a la lista de nodos que poseen una copia del bloque.

3. El nodo maestro realiza las siguientes acciones:

   (a) Envía una copia bloque al nodo local.

   (b) Envía una señal de ACK al nodo *home*. El nodo *home* actualiza la información de directorio y coloca como maestro al nodo local. Además, tanto si el estado original del bloque era *exclusive* como si era *master shared*, dicho estado pasa a *shared*.

   El estado del bloque en el nodo maestro pasa a *shared*, y el estado del bloque en el nodo local queda como *master shared*. Por lo tanto, al realizar una solicitud de lectura la propiedad del bloque se transfiere al nodo que ha solicitado la copia.

Solicitudes de escritura El protocolo interpreta los fallos de la cache en escritura como solicitudes de lectura en modo exclusivo. El nodo local deberá esperar a tener la copia en modo exclusivo para poder modificarla. Pueden darse tres casos diferentes:

- Que el bloque sobre el que se desea escribir esté en el nodo local en estado *shared* (figura 3.2(a)).
- Que el bloque esté en el nodo local en estado *shared* y que además el nodo local sea el nodo maestro (figura 3.2(b)).
- Que el bloque esté en estado *exclusive* en otro nodo (figura 3.2(c)).

Si la solicitud de lectura exclusiva se realiza sobre un bloque con copias en diferentes nodos, las operaciones que se realizan son las siguientes:

1. El nodo local envía la solicitud de lectura exclusiva al nodo que contiene el directorio *home*.

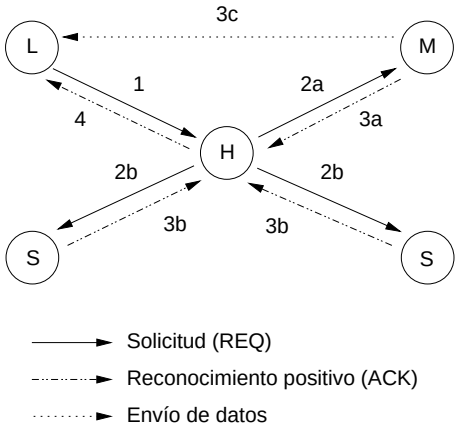
2. El nodo *home* realiza dos operaciones diferentes:

   (a) Pasa la solicitud al nodo que posee la copia maestra.

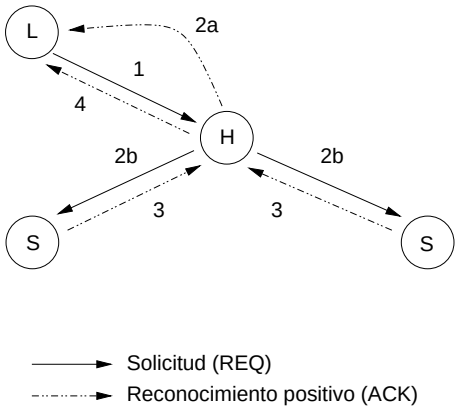
   (b) Envía mensajes de invalidación al resto de copias.

3. A continuación, se realizan las siguientes operaciones:

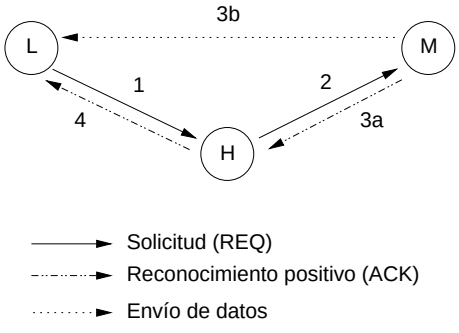
   (a) El nodo maestro envía un reconocimiento positivo (ACK) al nodo *home*, indicando que la operación puede completarse.



(a) Sobre un bloque *shared*



(b) Sobre un bloque *shared*, cuando el nodo local tiene la copia maestra



(c) Sobre un bloque *exclusive*

Figura 3.2: Solicitudes de escritura (“lectura exclusiva”) en COMA-F, según [45].

   (b) Los nodos que poseen copias del bloque envían reconocimientos positivos al nodo *home*. El estado de esos bloques en dichos nodos ha pasado a *Invalid*.

   (c) El nodo maestro envía el bloque al nodo local. El estado del bloque en el nodo maestro pasa a *Invalid*.

4. El nodo *home* cambia el estado del bloque a *exclusive* en la información de directorio. Además, envía un mensaje de reconocimiento positivo al nodo local. En el nodo local el estado del bloque pasa a *exclusive*.

Si el nodo local es además el nodo maestro, las operaciones que se realizan son las siguientes:

1. El nodo local envía la solicitud de lectura exclusiva al nodo *home*.
2. El nodo *home* realiza dos operaciones diferentes:
   (a) Devuelve la solicitud al nodo, ya que éste posee la copia maestra.
   (b) Envía mensajes de invalidación al resto de copias.
3. Los nodos que poseen copias del bloque envían reconocimientos positivos al nodo *home*. El estado de esos bloques ha pasado a *Invalid*.
4. El nodo *home* cambia el estado del bloque a *exclusive* en la información de directorio. Además, envía un mensaje de reconocimiento positivo al nodo local. En el nodo local el estado del bloque pasa a *exclusive*.

El tercer caso de solicitud de lectura exclusiva que nos queda por analizar se produce cuando el bloque solicitado está en estado exclusivo en otro nodo (figura 3.2(c)). Las operaciones a realizar en este caso serán las siguientes:

1. El nodo local envía la solicitud de lectura exclusiva al nodo *home*.
2. El nodo *home* pasa la solicitud al nodo maestro.
3. El nodo maestro realiza dos operaciones diferentes:
   (a) Envía un reconocimiento positivo al nodo *home*.
   (b) Envía el bloque de datos al nodo local.

   El estado del bloque en el nodo maestro pasa a *Invalid*.
4. El nodo *home* cambia el estado del bloque a *exclusive*. Además, envía un mensaje de reconocimiento positivo al nodo local. En el nodo local el estado del bloque pasa a *exclusive*.

El reemplazo en COMA-F

En COMA-F la cantidad de memoria máxima que puede utilizarse simultáneamente es igual a la suma de la capacidad de las diferentes memorias de atracción. Cuando un nodo de COMA-F necesita espacio para almacenar un bloque y no quedan marcos de bloque inválidos en su memoria de atracción, se procede al desalojo de un bloque.

Existen tres posibles casos de reemplazo en COMA-F, según la naturaleza del bloque a desalojar:

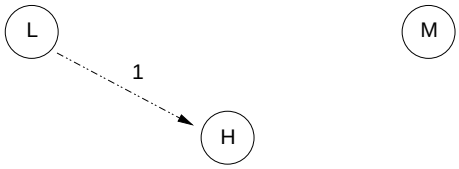
- Que el bloque a desalojar sea un bloque *shared* (figura 3.3(a)).
- Que el bloque a desalojar sea un bloque *master shared* con otras copias en el sistema (figura 3.3(b)).
- Que el bloque a desalojar sea un bloque *exclusive* o un *master shared* sin otras copias en el sistema. (figura 3.3(c)).

En el primer caso, tenemos un bloque *shared* ocupando el marco de bloque en donde deseamos almacenar nuestro bloque. El desalojo es sencillo: dado que existen otras copias del bloque en el sistema, basta con enviar una notificación al nodo que posee la información de directorio indicando que dicho bloque será sobreescrito. El estado del bloque pasa a *Invalid* en la memoria de atracción local, mientras que el nodo *home* elimina al nodo local de la lista de nodos que poseen ese bloque (figura 3.3(a)).

La segunda posibilidad que se presenta es que el bloque a desalojar sea la copia maestra, existiendo varias copias compartidas en el sistema. Dicho bloque también puede sobreescribirse, pero el nodo *home* debe designar a una de las copias restantes como copia maestra (figura 3.3(b)). Las operaciones a realizar son:

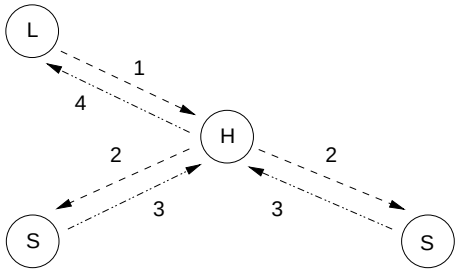
1. El nodo local (aquí, el nodo maestro) envía una solicitud de reemplazo maestro al nodo *home*, tras lo cual se modifica el estado del bloque en el nodo maestro a *Invalid*.
2. El nodo *home* cambia el puntero al nodo maestro, designando a uno de los nodos que poseen una copia como nuevo nodo maestro para ese bloque, y enviándole una solicitud de reemplazo maestro.
3. El nodo que recibe la solicitud cambia el estado del bloque a *master shared* y envía un reconocimiento positivo al nodo *home*.
4. El nodo *home* envía un reconocimiento positivo de reemplazo al nodo local.

Dado que en los dos casos anteriores existe al menos un nodo que posee una copia del dato, no ha sido necesario transferir el bloque desalojado a



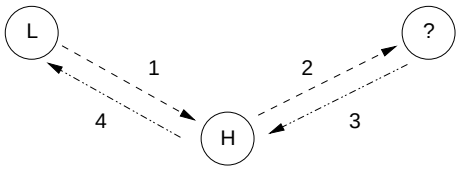
Reconocimiento positivo (ACK)

(a) Sobre un bloque *shared*



Solicitud de reemplazo
Reconocimiento positivo (ACK)
Envío de datos

(b) Sobre un bloque *master shared*



Solicitud de reemplazo
Reconocimiento positivo (ACK)

(c) Sobre un bloque *exclusive*, o un bloque *master shared* sin copias

Figura 3.3: Reemplazo en COMA-F, según [45].

ningún otro nodo. El tercer caso que puede presentarse es precisamente ese: que el bloque a desalojar sea *exclusive*, o bien *master shared* sin otras copias en el sistema. En este caso (figura 3.3(c)) las operaciones que se realizan son las siguientes:

1. El nodo local (aquí, el nodo maestro) envía una solicitud de reemplazo maestro al nodo *home*, tras lo cual se modifica el estado del bloque en el nodo maestro a *Invalid*.

2. El nodo *home* selecciona un nodo de forma aleatoria y le envía una solicitud de reemplazo exclusivo, junto con el bloque a reubicar.

3. El nodo seleccionado envía un reconocimiento positivo al nodo *home*, y coloca el estado del bloque como *exclusive*.

   Si el nodo seleccionado para que reciba el bloque no tiene espacio, envía un reconocimiento negativo de reemplazo al nodo *home*, el cual seleccionará a otro nodo, volviendo a intentar la operación, hasta que un nodo lo acepte.

4. El nodo *home* envía al nodo local un reconocimiento positivo de reemplazo.

Dado que la presión de memoria no llega al 100 %, el número total de marcos de bloque del sistema es menor que el número total de bloques. En consecuencia, se garantiza que el bloque desalojado podrá ser almacenado en algún nodo del sistema.

Solicitudes de lectura/escritura DMA y de borrado de bloques

Además de las operaciones vistas, el protocolo COMA-F tiene en cuenta la posibilidad de que un controlador DMA pueda acceder a la memoria de atracción de un nodo sin intervención del procesador. Para el caso de las lecturas DMA, los datos abandonan el marco de coherencia de la memoria y se almacenan en un dispositivo externo. En el caso de las escrituras, los datos se escriben en la memoria secuencialmente, respetando el mecanismo de coherencia. Describiremos brevemente el mecanismo utilizado.

Las lecturas y escrituras DMA sobre el espacio de memoria compartido debe hacerse de tal manera que se cumplan dos condiciones:

- Las lecturas DMA deben devolver el valor más reciente escrito en el bloque.

- Las escrituras DMA deben modificar todas las copias existentes del bloque de memoria correspondiente, con el objeto de mantener la consistencia de los datos.

El protocolo COMA-F realiza las lecturas y escrituras DMA con la premisa de que la información de directorio no debe verse modificada. Como veremos más adelante, cuando se produce una lectura DMA la lectura se hace sobre cualquiera de las copias del bloque que existen en el sistema, y la existencia de dichas copias no se ve perturbada por la lectura. Similarmente, cuando se produce una escritura DMA sobre un bloque, el propietario del bloque no cambia, por lo que no es necesario modificar la información de estado del protocolo de coherencia.

En el caso de las lecturas DMA, si un bloque no está presente en el nodo local se produce un fallo de lectura, que es tratado con el mecanismo ya descrito. Por lo tanto, aunque la lectura DMA de un bloque es idéntica a la lectura por parte del procesador, como veremos no ocurre lo mismo con las escrituras.

Dichas escrituras DMA pueden darse sobre tres tipos de bloque:

Sobre bloques inválidos : este caso se corresponde con la escritura sobre bloques inválidos *en todo el sistema*, aunque pertenecientes al espacio de direcciones compartido de la aplicación. En este caso, el nodo *home* (que es fijo para cada bloque, sea válido o no) deberá convertir esta solicitud en una solicitud de reemplazo exclusivo, inyectando el bloque en un nodo destino. Por lo tanto, cuando se realiza una escritura DMA sobre un bloque inválido, dicho bloque se inserta como un bloque nuevo.

Sobre bloques compartidos : en este caso, los bloques compartidos se actualizan con la nueva información. Por lo tanto, una escritura DMA sobre bloques compartidos no requiere el paso de dicho bloque al estado *exclusive*, sino que se utiliza un mecanismo de escritura actualizante para todas las copias.

Sobre bloques exclusivos : en este caso, el nodo que posee la copia maestra recibe el nuevo valor.

En consecuencia, cuando se produce una escritura DMA sobre un bloque inválido, dicho bloque se inserta en el sistema. Las lecturas DMA no pueden borrar automáticamente los bloques a los que acceden porque no está claro si ese bloque sigue siendo necesario en el sistema. Por lo tanto, la solución más conservadora es mantener el bloque en el sistema. En consecuencia, se necesita un mecanismo que permita invalidar (borrar) un bloque que ya no sea necesario.

La operación de borrado de un bloque en COMA-F debe borrar todas las copias de un bloque de memoria y además limpiar la entrada de directorio correspondiente. Existen tres posibilidades: que el bloque a borrar sea compartido por varios nodos (figura 3.4(a)), que sea compartido y además el nodo local sea el nodo maestro (figura 3.4(b)) y que el bloque sea exclusivo (figura 3.4(c)).

Si el bloque a borrar está compartido entre diferentes nodos, las operaciones a realizar son las siguientes (figura 3.4(a)):

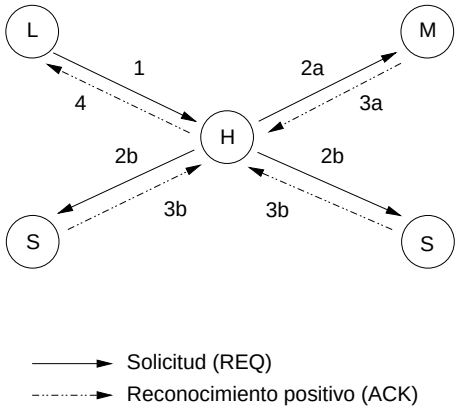
1. El nodo local envía la solicitud de borrado al nodo *home*.
2. El nodo *home* realiza dos operaciones diferentes:
   (a) Pasa la solicitud de borrado al nodo *master*.
   (b) Envía invalidaciones al resto de copias del bloque.
3. El nodo *home* recibe dos respuestas:
   (a) El nodo maestro le envía un reconocimiento positivo y coloca al bloque como inválido.
   (b) Los restantes nodos con copia del bloque envían reconocimientos positivos y colocan al bloque como inválido.
4. El nodo *home* cambia el estado del bloque a *Invalid* en la información de directorio. Además, envía un mensaje de reconocimiento positivo al nodo local. En el nodo local el estado del bloque pasa a *Invalid*.

Si la solicitud de borrado debe hacerse sobre un bloque compartido entre varios nodos y el nodo local es el nodo *master* para ese bloque, las operaciones a realizar son las siguientes (figura 3.4(b)):

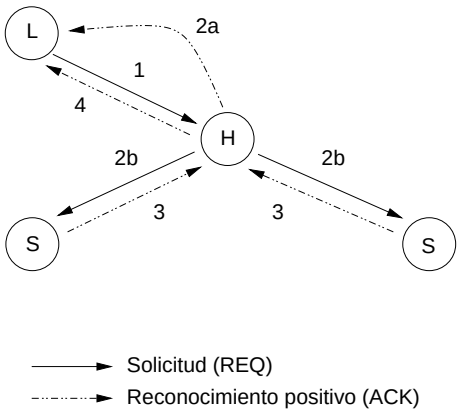
1. El nodo local envía la solicitud de borrado al nodo *home*.
2. El nodo *home* realiza dos operaciones diferentes:
   (a) Devuelve la solicitud al nodo, ya que éste posee la copia maestra.
   (b) Envía mensajes de invalidación al resto de copias.
3. Los nodos que poseen copias del bloque envían reconocimientos positivos al nodo *home*. El estado de esos bloques pasa a *Invalid*.
4. El nodo *home* cambia el estado del bloque a *Invalid* en la información de directorio. Además, envía un mensaje de reconocimiento positivo al nodo local. En el nodo local el estado del bloque pasa a *Invalid*.

Finalmente, nos queda el caso de que el bloque sea de uso exclusivo por parte de un nodo. Los mensajes a intercambiar en este caso son los siguientes (figura 3.4(c)):

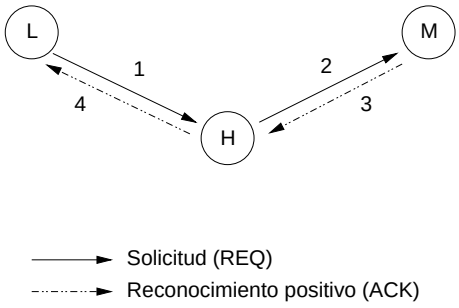
1. El nodo local envía la solicitud de borrado al nodo *home*.
2. El nodo *home* pasa la solicitud al nodo maestro.



(a) Sobre un bloque *shared*



(b) Sobre un bloque *master shared*



(c) Sobre un bloque *exclusive*

Figura 3.4: Borrado de un bloque en COMA-F, según [45].

3. El nodo maestro envía un reconocimiento positivo al nodo *home*. El estado del bloque en el nodo maestro pasa a *Invalid*.

4. El nodo *home* cambia el estado del bloque a *Invalid* en la información de directorio. Además, envía un mensaje de reconocimiento positivo al nodo local. En el nodo local el estado del bloque pasa a *Invalid*.

Sincronización

Las operaciones de sincronización permiten a los procesos gestionar el acceso a datos compartidos. COMA-F utiliza los mecanismos ya descritos para implementar las operaciones de sincronización. De esta manera se minimiza la complejidad hardware de dichas operaciones. COMA-F supone que el procesador utilizado en los nodos posee alguna instrucción atómica del tipo *test and set* (TAS), o bien un par de instrucciones del tipo *load linked* (LL) y *store conditional* (SC). Las instrucciones de tipo *load linked* obtienen un bloque de memoria de forma exclusiva, mientras que las instrucciones de tipo *store conditional* escriben el valor especificado a la dirección destino sólo si el bloque cargado con la instrucción LL no ha sido leído ni modificado por ninguna fuente externa.

El par de instrucciones LL/SC pueden utilizarse para acceder en escritura a través de una operación atómica de la siguiente manera: la instrucción LL obtiene un bloque de memoria en estado exclusivo, mientras que la instrucción SC escribe el valor especificado en la dirección destino del bloque cargado con LL sólo si el bloque no ha sido leído o modificado por una fuente externa. Si la instrucción SC no puede completarse debido a que el bloque ha sido accedido desde fuera, se devuelve una condición de error. Estas primitivas permiten construir otras primitivas de sincronización más complejas, como por ejemplo barreras.

Conclusiones

La utilización de directorios centralizados fijos presenta ventajas e inconvenientes. Entre sus principales ventajas podemos citar la independencia de la topología de conexión entre los nodos, ya que no se requiere una topología en bus común para establecer un mecanismo basado en directorios centralizados. Además, el almacenamiento en un nodo *home* de la lista de nodos que comparten un bloque concreto permite disminuir la sobrecarga de memoria asociada a la información de localización, ya que ésta no está duplicada en el sistema. Esto presenta la ventaja asociada de que se evitan los problemas de carreras, ya que se centralizan todas las solicitudes de modificaciones de bloques.

Uno de los mayores inconvenientes de la utilización de este mecanismo de localización de bloques consiste en que todas las transacciones que se

realizan en el sistema deben pasar obligatoriamente por el nodo *home*. Como además los nodos *home* se distribuyen estáticamente al principio de la ejecución de la aplicación, un nodo que sea *home* para un bloque determinado puede verse constantemente consultado por el estado de un bloque que ya no utiliza. Por lo tanto, aunque un bloque pueda migrar de un nodo a otro, su información de directorio permanece fija, lo que reduce en gran medida las ventajas de utilización de una arquitectura de tipo COMA.

Por otra parte, la independencia de la topología de interconexión que presenta COMA-F es un factor que influye negativamente en el rendimiento de este protocolo sobre un bus común. En efecto, para mantener dicha independencia, las transacciones en COMA-F no hacen uso de la capacidad de difusión del bus común, por lo que el número de mensajes que se intercambian en cada transacción es muy alto. Por ejemplo, si se desean invalidar n copias remotas de un bloque, el nodo *home* genera n mensajes, cuando en un bus con difusión bastaría con uno solo.

Respecto al mecanismo de reemplazo, cabe destacar que, pese a lo elaborado de las transacciones que se realizan, el criterio de selección de nodo destino es extremadamente simple: el nodo destino se selecciona aleatoriamente. Por lo tanto, el nodo *home* no utiliza ningún criterio basado en la información de la que dispone, sino que se limita a enviar aleatoriamente solicitudes de reemplazo hasta que alguna es aceptada. En consecuencia, a presiones de memoria altas, la probabilidad de que una solicitud sea rechazada no es fija, sino que crece con el número de nodos, ya que al aumentar dicho número es menos probable que el nodo *home* acierte a enviar la solicitud de reemplazo al nodo que posee espacio libre para recibir el bloque. Como veremos en el capítulo 6, esta estrategia de selección del nodo destino influye negativamente en el rendimiento de un sistema COMA.

3.3.2 Gestión distribuida: el sistema DICE

DICE (Direct Interconnect of Computer Elements) es un proyecto de la Universidad de Minnesotta [57, 54, 41, 15, 56] que pretende desarrollar una arquitectura COMA eficiente en un bus común con procesadores simétricos. No se brinda una especial atención a la escalabilidad del sistema. En su lugar, se hace hincapié en disminuir la contención del bus y el ancho de banda necesario, reduciendo al mínimo el tráfico en el bus. Por lo tanto, el objetivo de este proyecto se centra más en el estudio de protocolos de coherencia y reemplazo que en el desarrollo de nuevas arquitecturas para cada nodo de proceso [15, 41].

Un nodo de proceso en DICE está formado por un microprocesador de alto rendimiento, dos niveles de cache y una memoria local que actúa como memoria de atracción. Las etiquetas de la memoria local, almacenadas en SRAM, están duplicadas para evitar conflictos entre los accesos locales y los del bus. A diferencia de COMA-F, en DICE se mantiene la propiedad

de subconjunto o inclusión en toda la jerarquía de memoria.

El protocolo de coherencia DICE [15, 56] describe el funcionamiento de un sistema COMA apto para su implementación en bus común. Para el mantenimiento de la coherencia, DICE aprovecha la característica de difusión que ofrece el bus común, utilizando un mecanismo de tipo *snoopy*. El sistema DICE utiliza un protocolo de coherencia de cuatro estados con escritura invalidante. La principal novedad que ofrece DICE respecto de otros protocolos de tipo COMA es el mecanismo de reemplazo utilizado. Como veremos más adelante, cuando un nodo desea desalojar un bloque, envía una solicitud a los restantes nodos. Al recibir dicha solicitud, cada nodo responde enviando a su vez un vector de prioridad, que permite a un sistema de arbitraje decidir el destino más apropiado para el bloque a desalojar.

Pese a que el funcionamiento del protocolo no depende de ningún bus en concreto, existe un diseño basado en el bus Futurebus+ que demuestra que puede construirse una máquina multiprocesador COMA basada en DICE utilizando componentes comerciales y un pequeño hardware adicional [56].

Protocolo de coherencia

La figura 3.5 muestra el diagrama de estados del protocolo de coherencia DICE. Dicho protocolo se basa en cuatro estados básicos: *INV* (la información presente en el bloque no es válida), *SHN* (la información del bloque es válida, el nodo no es propietario del bloque y existen otras copias en el sistema), *SHO* (el nodo es propietario del bloque y tal vez existan otras copias), y *EXL* (el nodo es propietario de la única copia del bloque). Los propietarios de cada bloque tienen la obligación de responder a las solicitudes respecto de dichos bloques.

Los eventos del protocolo entre nodos son los siguientes: *NR* (solicitud de lectura), *NW* (solicitud de escritura) y *NI* (invalidación). El evento *NR* se utiliza para el envío de copias de un bloque, al objeto de satisfacer solicitudes de lectura. El evento *NW* permite simultáneamente enviar el bloque y transferir su propiedad, para que el nodo que lo ha solicitado pueda realizar escrituras sobre él. El evento *NI* invalida las copias *SHN* de un bloque determinado en el resto de nodos. Para poder enviar este evento, un nodo deberá poseer una copia *SHO* o *SHN* del bloque. Una vez enviado, el bloque pasa automáticamente al estado *EXL*. Es interesante destacar que, dado que no se producen solapamientos entre diferentes transacciones en el bus, no cabe la posibilidad de que se produzcan condiciones de carrera asociadas a la solicitud de invalidación simultánea de un mismo bloque por parte de dos nodos.

Existen dos eventos de protocolo asociados al reemplazo: *NTO* (transferencia de propiedad) y *NNOC* (transferencia de la última copia de un

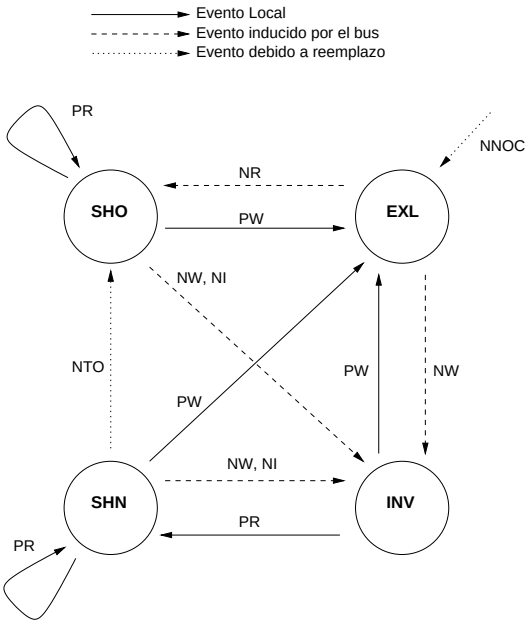


Figura 3.5: Protocolo de coherencia de DICE, según [15].

bloque). El primero se utiliza cuando existe al menos una copia *SHN* del bloque en el sistema. En este caso, el nodo propietario transfiere a dicho nodo la propiedad del bloque. El segundo se utiliza para la operación de reemplazo, transfiriendo la última copia de un bloque a un nodo remoto. Trataremos con detalle el reemplazo en DICE en la sección 3.3.2. Como puede verse en la figura 3.5, no se utilizan eventos de respuesta a los diferentes eventos de solicitud descritos.

En DICE, la utilización del concepto de propiedad determina de forma unívoca el nodo que debe responder a cada una de las solicitudes. Como veremos en la siguiente sección, el protocolo de coherencia basado en bus de DICE está relacionado íntimamente con el mecanismo de reemplazo utilizado.

El reemplazo en DICE

El problema de la aceptación de un bloque entrante en DICE tiene dos partes. En primer lugar, debe decidirse el marco de bloque en donde se almacenará el nuevo bloque. Suponiendo que la AM es asociativa por conjuntos, el primer candidato de entre todos los marcos de bloque del conjunto se elige entre los que posean estado *INV*. Si no existe ninguno, se sobreescribe un bloque que se encuentre presente en *SHN*, ya que se garantiza que existe al menos otra copia de dicho bloque en el sistema. Si esto no es posi-

ble, significa que todos los bloques presentes en el conjunto son propiedad del nodo local. Si existe algún bloque en estado *SHO* dentro del conjunto, y existe alguna copia *SHN* de dicho bloque en el sistema, se transfiere la propiedad de ese bloque a dicho nodo. Esto permite enviar el bloque a un nodo que lo haya estado utilizando en lugar de enviárselo a un nodo que no lo necesita. El problema en este caso es que el nodo local no sabe si existen copias *SHN* de su bloque *SHO*, ya que dichas copias pueden haber sido invalidadas localmente. Por otra parte, la relocalización de un bloque *EXL* supone transferir la única copia del bloque presente en el sistema.

Para resolver ambas cuestiones, DICE utiliza un esquema basado en prioridades. Dicho esquema permite asignar un nivel de prioridad a cada nodo remoto para la recepción del bloque, basado en el estado del conjunto correspondiente de cada AM remota.

Cuando el nodo local decide desalojar un bloque, envía una solicitud al bus. Todos los nodos remotos responden a dicha solicitud indicando su disponibilidad para aceptar el bloque, a través de un vector de prioridad que permite especificar cuatro posibilidades distintas. De mayor a menor prioridad, dichas posibilidades son las siguientes:

1. El nodo remoto posee una copia del bloque en estado *SHN*.
2. El nodo remoto posee al menos un marco de bloque en estado *INV*.
3. El nodo remoto posee al menos un marco de bloque en estado *SHN*.
4. El nodo remoto posee todos sus marcos de bloque en estado *SHO* o *EXL*.

Como se ha dicho anteriormente, la prioridad más alta se asigna al nodo que posea una copia del bloque que se pretende desalojar. El nivel siguiente de prioridad se reserva para aquellos nodos que no necesiten invalidar un bloque para aceptar el bloque entrante. De esta manera no se reduce la replicación en las memorias de atracción. El estado al que pasará el bloque una vez reemplazado será *EXL*, ya que si se envía el bloque a un nodo con prioridad 2 significa que no existen otras copias del bloque en el sistema.

El nivel de prioridad 3 se reserva para aquellos nodos que no posean marcos de bloque inválidos pero puedan invalidar bloques en estado *SHN* para aceptar el bloque entrante.

La prioridad más baja, finalmente, se reserva para nodos que sean propietarios de todos los bloques presentes en el conjunto correspondiente de su memoria de atracción. Este caso es muy poco frecuente [41], ya que indicaría una presión de memoria anormalmente alta. Sin embargo, el protocolo de reemplazo utilizado en DICE lo tiene en cuenta, utilizando en este caso un mecanismo de *intercambio*, consistente en enviar el bloque al nodo propietario del bloque que se pretende acomodar en su lugar en el

nodo local. El nodo propietario de dicho bloque responde a su vez enviando el bloque deseado. De esta manera, la propiedad de ambos bloques se intercambia, evitándose la generación de una cadena de reemplazos.

Para efectuar el desalojo, el nodo local coloca el bloque en un buffer externo a la AM, llamado "buffer de reubicación". Cuando todos los nodos envían su vector de prioridad a través del bus, un módulo de arbitraje se encarga de determinar el nodo destino. En ese momento, el nodo destino copia el bloque en un buffer local, denominado "buffer de entrada", desde donde pasa a la AM.

Como puede verse, el reemplazo en un sistema DICE compuesto de n nodos supone el envío de varios eventos:

- Un evento *NNOC* para solicitar el desalojo.
- $n-1$ respuestas, consistentes en el vector de prioridad de cada nodo remoto para el conjunto correspondiente de sus AM.
- Un evento proveniente del módulo de arbitraje indicando el nodo seleccionado.
- La transferencia del bloque desde el buffer de reubicación del nodo local al buffer de entrada del nodo destino.

Según la nomenclatura utilizada en DICE, el único evento propiamente dicho sería el evento *NNOC* enviado en primer lugar, mientras que el resto son simplemente transferencias realizadas a través del bus común. Esto se debe a que DICE utiliza como base un sistema fuertemente acoplado y un mecanismo de tipo *snoopy* que no requiere respuestas tras el envío de cada evento, al no poder solaparse las diferentes transacciones. Sin embargo, si consideramos como evento cada *transferencia* de datos a través del bus, podemos ver que el mecanismo de reemplazo en DICE comporta el envío de $n+2$ eventos para su finalización: una solicitud, $n-1$ respuestas, un evento de selección del nodo destino y un evento que incluye la transferencia del bloque.

En el caso de que sea necesario enviar el bloque a un nodo con prioridad 4, el número de transferencias a realizar es ligeramente diferente, ya que tras la selección del nodo destino y la aceptación por parte de éste del bloque entrante, dicho nodo deberá enviar a su vez al nodo local el bloque que éste necesita. Por lo tanto, se produce un evento adicional, quedando el número de eventos en $n+3$.

Conclusiones

La utilización de un mecanismo tipo *snoopy* en DICE permite aprovechar la característica de difusión del bus común, a diferencia de lo que ocurre en

COMA-F. Esto se consigue, sin embargo, a costa de impedir la utilización de otras redes de interconexión. Dado que DICE es un protocolo orientado a su utilización en sistemas fuertemente acoplados, impone algunas restricciones al bus, como por ejemplo la posibilidad de reserva del bus hasta que la transacción se complete. Esto evita la necesidad de utilizar estados transitorios, simplificándose en gran medida el funcionamiento del protocolo.

El mecanismo de reemplazo utilizado en DICE, basado en consulta, permite al nodo que inicia la operación de desalojo enviar el bloque al nodo más apropiado para recibirlo. Sin embargo, esto se consigue a través de un mecanismo de consulta, lo que incrementa el tráfico del bus. En efecto, dado un sistema de n nodos, el número de mensajes a intercambiar es de $n+2$ en el caso de que existan nodos con prioridades 1, 2 o 3, y de $n+3$ si todos los nodos tienen prioridad 4. Por lo tanto, la operación de reemplazo genera un número de mensajes que es $O(n)$. Como veremos en el capítulo 6, si la presión de memoria es alta, la generación de este número de mensajes aumenta sensiblemente el tráfico en el bus, disminuyendo el rendimiento del sistema.

3.3.3 Otros protocolos

En esta sección examinaremos brevemente otros protocolos COMA que permiten su implementación sobre un sistema de bus común, analizando las ventajas e inconvenientes de cada solución particular al problema del reemplazo.

DDM

El sistema DDM (Data Diffusion Machine) [34, 35], una de las primeras arquitecturas COMA, fue desarrollado en el SICS (Swedish Institute of Computer Science). Se propusieron dos versiones: un sistema DDM en bus único, y un sistema DDM que utiliza una jerarquía de buses. El sistema DDM en bus único utiliza un sistema de localización de bloques *snoopy* sobre un bus del tipo *split-transaction*, es decir, que permite el solapamiento de transacciones. Este hecho obliga a la utilización de estados transitorios en el protocolo [33, 32]. El protocolo de coherencia está formado por los siguientes estados: *Invalid*, *Exclusive* y *Shared*, más los estados transitorios *Reading* (se ha enviado una solicitud de lectura y se espera la respuesta), *Answering* (se está respondiendo una solicitud de lectura), *Waiting* (se ha enviado una solicitud de acceso exclusivo y se espera que se complete), y *Reading and Waiting* (se ha solicitado un bloque para luego modificar su valor).

Para resolver las solicitudes simultáneas sobre un bloque y las condiciones de carrera, DDM define un protocolo externo denominado *top protocol*. La utilización de este protocolo permite, por ejemplo, evitar que sean descartadas todas las copias *Shared* de un bloque.

En su versión jerárquica, el *top protocol* se reemplaza por un directorio centralizado, que permite decidir si una solicitud concreta puede resolverse en el mismo bus en el que se ha generado o hay que transferirla a la jerarquía superior, en lo que se denomina "lecturas y escrituras multinivel" [32].

El protocolo DDM incorpora mecanismos de reemplazo, tanto en su versión de bus único como en su versión jerárquica. Para seleccionar el bloque a desalojar, se elige en primer lugar al bloque en estado *Shared* que lleve más tiempo en la memoria de atracción local. Dado que el protocolo DDM no utiliza el concepto de "copia maestra", no puede descartarse un bloque *Shared* sin generar tráfico de red, ya que esto puede llevar a que se descarten simultáneamente todas las copias de un bloque. Por lo tanto, la selección de un bloque *Shared* conlleva la generación de un evento denominado *Out*. El *top protocol* verifica que queden copias en alguna de las restantes memorias de atracción: de no ser así, convierte el evento *Out* en un evento *Inject*, que se encarga de transferir la última copia de un bloque a otro nodo.

El mecanismo de selección de nodo destino de DDM es simple: en primer lugar, se intenta desalojar el bloque hacia un nodo que posea algún marco de bloque en estado *Invalid*. De no existir ninguno, se desaloja hacia un nodo con algún marco *Shared*. Según las especificaciones de DDM [32], este sistema garantiza que se encontrará un nodo que acepte el bloque, ya que la presión de memoria será siempre menor que el 100%.

La utilización de mecanismos de tipo *snoopy* en DDM reduce el número de eventos necesarios para el mantenimiento de la coherencia. Por otra parte, la utilización de un bus que permite solapar transacciones minimiza los requisitos que debe cumplir la red de interconexión, a costa de complicar el diseño del protocolo con la utilización de estados transitorios. Sin embargo, la ausencia de un mecanismo de identificación de la copia maestra de un bloque compartido obliga a la utilización de un protocolo externo que impida la eliminación de la última copia del bloque. Este protocolo deberá llevar la cuenta de las copias restantes de cada bloque del espacio de trabajo, autorizando o no su eliminación. En una organización jerárquica, la utilización de dicho protocolo bajo la forma de un mecanismo de directorio permite encaminar eficazmente las solicitudes hacia la subred que contenga el bloque buscado. Sin embargo, la utilización de un protocolo de este tipo en una organización de bus único complica el funcionamiento del sistema y centraliza las decisiones, no pudiéndose realizar una gestión distribuida de la coherencia pese a utilizarse mecanismos *snoopy* sobre bus común.

Respecto al mecanismo de selección de nodo destino en una operación de reemplazo, la estrategia propuesta por DDM no contempla algunos casos que pueden darse en este sistema a presiones de memoria altas. Por ejemplo, puede darse la circunstancia de que en el instante de desalojar un bloque, todos los marcos de bloque de los conjuntos afectados se encuen-

tren simultáneamente en estados transitorios. En este caso, la estrategia de DDM consistente en buscar nodos con marcos *Invalid* o *Shared* fracasaría.

Bus-based COMA

El proyecto DDM ha dado lugar a diferentes arquitecturas relacionadas: cabe destacar BB-COMA [51], un sistema COMA en bus común creado a partir de la arquitectura DDM no jerárquica. Entre sus principales novedades, BB-COMA incorpora el estado estable *Owner*, utilizado para designar la copia maestra de un bloque. De esta manera se elimina la necesidad de un protocolo externo al protocolo de coherencia para evitar la eliminación de la última copia de un bloque.

La selección del nodo destino de una operación de desalojo se realiza a través de un mecanismo basado en prioridades, idea que como vimos en la sección 3.3.2 ha sido adoptada en DICE. Cuando un nodo desea desalojar un bloque del que es propietario, envía un evento de desalojo que es recibido por todos los nodos del sistema. Cada uno de los nodos responde con un vector de prioridad de dos bits, similar al utilizado en DICE, tras lo cual se selecciona el nodo destino. A presiones de memoria menores que el 100%, se garantiza que siempre habrá un nodo dispuesto a aceptar el bloque. Este sistema elimina la necesidad de un mecanismo externo de gestión del reemplazo, pero adolece del mismo problema visto en DICE: una solicitud de reemplazo genera un número de eventos que crece linealmente con el número de nodos.

COMA-BC

El protocolo COMA-BC [85, 84, 86] es un protocolo COMA sobre bus común de tipo híbrido, con directorios y mecanismo *snoopy*. Su diseño se orienta a minimizar la necesidad de hardware específico, lo que permite la reducción del coste asociado a las máquinas COMA. Esta idea es la que subyace en los protocolos de tipo “COMA software” [51, 52].

Cada uno de los bloques de la memoria compartida en COMA-BC lleva asociado un único propietario, que es el nodo encargado de responder a las solicitudes de lectura o escritura provenientes de otros nodos. El propietario de un bloque no es fijo: cuando un nodo solicita un bloque para escribir sobre él, dicho nodo pasa a ser el nuevo propietario del bloque.

En COMA-BC cada nodo dispone de un controlador de coherencia que se encarga de gestionar la información de control presente en la memoria de atracción. Para ello, el controlador de coherencia del nodo dispone de una tabla con la información de estado y etiqueta referente a cada marco de bloque de la memoria de atracción local. Además, dicho controlador mantiene de forma local una tabla con la información de directorio, en donde

se almacena la identidad del propietario de cada bloque presente en el sistema.

COMA-BC se diseñó al objeto de que el número de eventos que circulan por el bus se mantenga lo más bajo posible, al ser el bus el cuello de botella de esta clase de sistemas. Para ello, se aprovecha la capacidad de difusión que poseen los sistemas basados en bus común, mediante la cual cada mensaje enviado por un nodo es recibido por todos los demás. COMA-BC utiliza esta característica para mantener actualizada la información de directorio y estado de cada nodo.

En COMA-BC existen tres diferentes estados posibles para cada bloque: *Inválido*, que indica que el contenido del marco cache no es válido; *Limpio*, que indica que puede accederse al bloque para lectura; y *Sucio*, que indica que puede accederse al bloque tanto para lectura como para escritura. Por otra parte, existen dos estados transitorios: *Inv-Esp-RRB* e *Inv-Esp-RRI*. Estos estados permiten indicar que el contenido del marco correspondiente es inválido, y que dicho marco está pendiente de la resolución de una operación de lectura o escritura, respectivamente. La necesidad de estados transitorios es una consecuencia de la utilización de un bus común en el que pueden solaparse transacciones. En efecto, al no existir la posibilidad de reservar el bus hasta que se complete una transacción, dichas transacciones pueden solaparse, iniciándose una segunda antes de que se complete la primera. Esta característica obliga a la utilización de estados transitorios, para indicar la existencia de una transacción en curso sobre el bloque correspondiente.

3.4 Nuestra propuesta: VSR-COMA

En los protocolos COMA examinados destacan dos mecanismos de selección del nodo destino: la selección aleatoria, utilizada en COMA-F, y la selección por consulta, utilizada en BB-COMA y en DICE. Si partimos de la base de que la información presente en cada nodo es insuficiente para decidir el nodo destino más apropiado en una operación de reemplazo, estas dos soluciones parecen las más lógicas: o se elige un nodo destino al azar, o se consulta a todos los nodos sobre su disponibilidad.

La solución que proponemos en el presente trabajo parte de un enfoque diferente. La utilización de la característica de difusión del bus común permite a todos los nodos recibir todos los eventos intercambiados en el sistema. Por lo tanto, si suponemos que todos los nodos conocen el estado inicial del sistema, es teóricamente posible para cada nodo seguir la evolución de las memorias de atracción del sistema sin más que procesar todos los eventos que reciba, llevando la cuenta del estado de todos los marcos de bloque de las memorias de atracción remotas. Como es natural, el almacenamiento de toda esa información de estado supone una sobrecarga en

la memoria de control utilizada por cada controlador de coherencia. Dicho esfuerzo de almacenamiento adicional nos permite a cambio realizar una selección óptima del nodo destino de una operación de desalojo basándonos en el estado de todo el sistema, sin necesidad de generar un solo evento adicional. La cuestión consiste en determinar la importancia relativa entre el ahorro de eventos en la red y la sobrecarga de memoria asociada a esta solución, ya que para sistemas multicomputador basados en bus común la resolución de una falta remota puede suponer un tiempo varios órdenes de magnitud superior al acceso en caso de acierto. El protocolo VSR-COMA (Valladolid Smart Replacement COMA), cuya descripción y análisis de rendimiento nos ocupará en los capítulos siguientes, utiliza este enfoque.

3.5 Conclusiones

En el presente capítulo se ha discutido el problema del reemplazo en los protocolos COMA en bus común, examinando las soluciones propuestas hasta la fecha y evaluando sus ventajas e inconvenientes. Como se ha visto, el problema de la selección del nodo destino de una operación de desalojo no se ha resuelto aún satisfactoriamente: tanto la selección aleatoria del nodo destino (utilizada en COMA-F) como la selección basada en consulta (utilizada en DICE) presentan características que perjudican su comportamiento a presiones de memoria altas.

En el siguiente capítulo describiremos VSR-COMA, un protocolo COMA híbrido en bus común en el que cada nodo de proceso sigue la evolución de las caches remotas, al objeto de poder seleccionar el nodo destino de una operación de desalojo sin necesidad de generar tráfico adicional.

Capítulo 4

El protocolo VSR-COMA

4.1 Introducción

El protocolo VSR-COMA [63] supone una evolución del protocolo COMA-BC, descrito en la sección 3.3.3. COMA-BC es un protocolo COMA en bus común que busca obtener buenas prestaciones mediante la utilización de múltiples computadores de propósito general.

El rendimiento de COMA-BC se ha evaluado ejecutando aplicaciones pertenecientes al conjunto Splash-2 [11], obteniéndose índices comparables a otras arquitecturas COMA basadas en redes de estaciones de trabajo [84], lo que demuestra la viabilidad de un protocolo COMA de gestión distribuida sobre bus común. Sin embargo, COMA-BC tiene algunas limitaciones. La principal atañe a la ausencia de un mecanismo de desalojo de bloques. Esto obliga a que el tamaño de la memoria de atracción local en cada nodo sea igual al tamaño del espacio compartido. Aunque este hecho favorece en gran medida la replicación de bloques en el sistema, limita la escalabilidad del tamaño del espacio de direcciones compartido de las aplicaciones. En segundo lugar, como una consecuencia de la limitación descrita, la función de mapa de la memoria de atracción local posee correspondencia directa, ya que no tiene sentido un mecanismo de tipo asociativo por conjuntos al no poder desalojarse bloques. En tercer lugar, sería deseable que el protocolo incorporara operaciones de memoria del tipo *Test & Set* (TAS) y *Fetch & Increment* (FAI), para implementar mecanismos de sincronización entre procesos. El desarrollo de VSR-COMA busca eliminar estas limitaciones de COMA-BC.

A continuación se enumerarán más detalladamente los objetivos de diseño de VSR-COMA y se expondrá el funcionamiento del protocolo.

4.2 Objetivos de diseño

Los objetivos de diseño de VSR-COMA se basan en la obtención de un protocolo de coherencia tipo COMA con reemplazo para un sistema multicomputador basado en un conjunto de estaciones de trabajo conectadas a través de un bus común. La eliminación de las limitaciones de COMA-BC descritas en el apartado anterior se traducen en los siguientes requisitos:

- Desarrollo de un mecanismo de reemplazo que utilice la característica de difusión del bus común para seleccionar el nodo destino más apropiado para el desalojo del bloque.
- Utilización en la memoria de atracción de una función de mapa de tipo asociativo por conjuntos.
- Implementación de operaciones de memoria de tipo *Test & Set* (TAS) y *Fetch & Increment* (FAI), al objeto de facilitar la utilización de mecanismos de sincronización entre procesos.

En el desarrollo de VSR-COMA se ha considerado de especial importancia la reducción al mínimo del tráfico generado en la red. Esto ha llevado a la necesidad de establecer un mecanismo de reemplazo que seleccione el nodo destino más apropiado para recibir el bloque. Como hemos visto en el capítulo 3 del presente trabajo, la selección del nodo destino en otros protocolos COMA de gestión distribuida se realiza bien de forma aleatoria (como es el caso de COMA-F [45]) o bien utilizando mecanismos de consulta para establecer el nodo con mejores posibilidades de aceptar el bloque, solución adoptada en DICE [15, 56].

Ambas soluciones generan tráfico adicional en la red. En el caso de la selección aleatoria del nodo destino, la solicitud de reemplazo utiliza sólo un evento de red, lo que es el mínimo imprescindible. Sin embargo, las probabilidades de que el nodo así elegido no disponga de espacio libre provoca que la solicitud de reemplazo pueda ser rechazada, generándose una nueva solicitud e incrementando el tráfico de red. Dicha probabilidad aumenta con la presión de memoria y con el número de nodos. Por otra parte, aunque la solicitud sea aceptada, cabe la posibilidad de que exista otro nodo que haya utilizado el bloque en el pasado y que esté en mejores condiciones de admitirlo, disminuyéndose así la tasa de fallos en el futuro.

En el caso de la selección de nodo por consulta, el número de eventos a intercambiar es alto. Como hemos visto en el capítulo 3, dados n nodos, el número de eventos que se generan para determinar el nodo destino es de al menos $n + 2$, divididos como sigue: una solicitud, $n - 1$ respuestas, la selección del nodo destino y el envío del bloque. Para seleccionar el nodo destino, DICE utiliza un mecanismo basado en prioridades, en donde cada nodo evalúa su capacidad para recibir el bloque y responde a la solicitud

de desalojo indicando su prioridad. Este mecanismo es más elaborado que el de selección aleatoria, ya que utiliza la información de estado de cada nodo para seleccionar el destino más adecuado. Sin embargo, esto se realiza a expensas de un tráfico de red mucho mayor, tráfico que además crece linealmente con el número de nodos.

El protocolo VSR-COMA realiza la selección del nodo destino de una operación de reemplazo en función del estado de las memorias de atracción de los nodos remotos. Sin embargo, contrariamente a lo que sucede en DICE, esto no supone ningún incremento del tráfico de red, ya que cada nodo en VSR-COMA utiliza la información intercambiada a través del bus para mantener actualizada una estructura de datos que contiene el estado de cada una de las memorias de atracción remotas. De esta manera, cuando se necesita desalojar un bloque, el nodo local puede seleccionar el nodo remoto más apropiado para recibir dicho bloque. Por otra parte, la presencia en cada nodo de toda la información de reemplazo disponible permite utilizar estrategias de reemplazo más elaboradas, sin que esto suponga un aumento de tráfico en la red.

El protocolo VSR-COMA, cuyo funcionamiento se expondrá en este capítulo, ha sido desarrollado y validado utilizando redes de Petri coloreadas. El proceso de validación del funcionamiento ha resultado especialmente complejo, ya que el mantenimiento de la información de control implica un gran número de acciones a realizar por cada controlador de coherencia. El apéndice A describe en detalle el mecanismo de modelado utilizado.

4.3 Gestión distribuida de la coherencia

La figura 4.1 muestra la estructura de un nodo de proceso en VSR-COMA. El controlador de coherencia tiene dos funciones:

- Responder a las solicitudes por parte del procesador, al objeto de efectuar operaciones de memoria sobre los datos presentes en la memoria de atracción. La comunicación entre el procesador y el controlador de coherencia se realiza a través del *protocolo de memoria.*

- Interactuar con los controladores de coherencia presentes en el resto de nodos del sistema, con el fin de mantener coherentes los datos presentes en las diferentes memorias de atracción. La comunicación entre los controladores a través de la red se realiza a través del *protocolo entre nodos.*

Con este doble fin, cada controlador de coherencia mantiene tres estructuras de datos claramente diferenciadas:

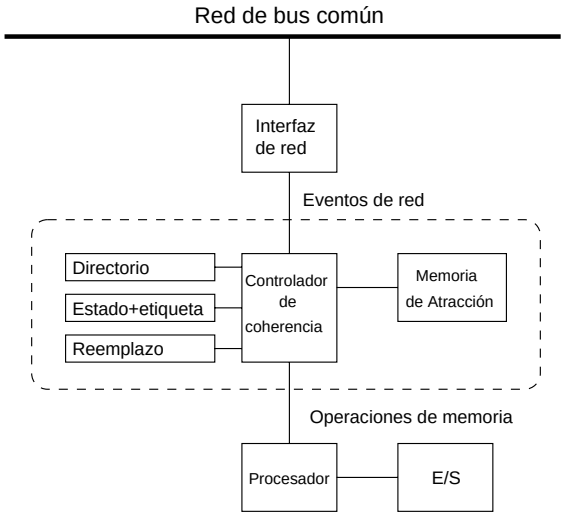


Figura 4.1: Estructura de un nodo de proceso en VSR-COMA.

Directorio : la información de directorio consiste en una tabla que indica el nodo propietario de cada uno de los bloques que componen el espacio de direcciones compartido. Esta información permite al controlador de coherencia dirigirse al propietario del bloque ante cualquier operación de memoria relativa a dicho bloque.

Estado+etiqueta : la información de estado y etiqueta almacena el estado y la etiqueta asociada a cada uno de los marcos de bloque de los conjuntos que componen la memoria de atracción local. Esta información es análoga a la existente en cualquier cache con estructura asociativa por conjuntos.

Reemplazo : la información de reemplazo constituye una de las principales novedades del protocolo VSR-COMA. Se trata de una tabla que contiene la información de estado y etiqueta de cada uno de los marcos de bloques de los conjuntos que componen las memorias de atracción *remotas*. Por lo tanto, consultando esta tabla es posible conocer la situación de un conjunto determinado en todas las memorias de atracción remotas. Esta información, como se verá más adelante, resulta muy útil para decidir, entre otras cosas, el nodo destino en una operación de reemplazo.

La actualización de las tres estructuras de datos es tarea del controlador de coherencia local, aprovechando la circunstancia de que todos los controladores de coherencia reciben todos los eventos de red, aunque sólo el destinatario de dicho evento está obligado a responder. Así, todos

los eventos intercambiados entre los diferentes controladores de coherencia llevan un identificador de "nodo destino". Este identificador permite asegurar que cada evento será tratado y respondido por exactamente un nodo. Podría pensarse que, dado que todos los nodos están unidos por un bus común, no resulta necesario especificar el nodo destino de una solicitud de lectura o escritura, ya que cada nodo sabe en cada momento si es o no el propietario de un bloque. Sin embargo, si el control de acceso al bus se realiza a través de un mecanismo de contienda, puede darse el caso de que una solicitud sea enviada en el momento en que el bloque correspondiente está cambiando de propietario, por lo que ningún nodo responderá a la solicitud. Este tipo de condiciones de carrera son frecuentes, ya que el protocolo permite que se solapen pares de evento solicitud-respuesta, al objeto de mejorar las prestaciones.

VSR-COMA utiliza el concepto de *propiedad* para cada uno de los bloques presentes en el sistema. Este concepto es análogo al utilizado en COMA-BC. Todos los bloques tienen un propietario, que es el encargado de responder a las solicitudes de lectura o escritura que el resto de nodos realicen sobre ese bloque. La propiedad de un bloque cambia dinámicamente, a medida que los nodos realizan operaciones de escritura sobre dicho bloque.

A grandes rasgos, el funcionamiento del protocolo puede describirse como sigue:

- Si se produce una solicitud de lectura, el propietario responde enviando una copia del bloque al nodo que lo ha solicitado.
- En caso de que la solicitud sea de escritura, el nodo propietario también envía una copia del bloque, pero el nodo que ha realizado la solicitud pasa a ser el nuevo propietario, invalidándose el resto de copias en los demás nodos.
- Si un nodo desea desalojar un bloque del que es propietario, dicho nodo deberá seleccionar un nodo destino y enviarle una solicitud de desalojo. El nodo destino decidirá si acepta el bloque en función del estado de su memoria de atracción. Si dispone del espacio necesario para admitir el bloque, el nodo destino pasa a ser el nuevo propietario. En caso contrario, se envía una respuesta negativa al nodo que ha hecho la solicitud, el cual deberá seleccionar un nuevo nodo destino.

Como se ha dicho arriba, cuando un nodo solicita un bloque en exclusiva pasa a ser el nuevo propietario del bloque. En consecuencia, todos los controladores de coherencia actualizan la información de directorio, almacenando el identificador del nuevo propietario. La actualización de la información de estado y etiqueta es sencilla, al depender sólo de las operaciones de memoria solicitadas por el nodo local y de las eventuales solicitudes remotas que se reciban. El mantenimiento de la información de reemplazo,

sin embargo, requiere que cada controlador de coherencia analice la información presente en todos los eventos intercambiados. Volveremos sobre este problema en la sección 4.5, una vez vista la nomenclatura utilizada en la descripción del protocolo.

4.4 Nomenclatura

4.4.1 Protocolo de memoria

Se denomina *operación de memoria* a cada solicitud de lectura, escritura, TAS o FAI realizada por el procesador y dirigida a un bloque de la memoria compartida. En VSR-COMA, el modelo de consistencia utilizado es el de consistencia secuencial [29, 95], no considerándose necesaria la implementación de modelos de consistencia más relajados [37]. Por lo tanto, hasta que el controlador de coherencia no devuelve el resultado de una operación de memoria, el procesador no puede solicitar la siguiente. Las respuestas a las operaciones son siempre afirmativas, recibiendo el nombre de *notificaciones*. Las operaciones de memoria implementadas y sus correspondientes notificaciones son las siguientes:

PrRd (*Processor Read*): solicitud de lectura de un dato del espacio compartido de direcciones por parte del procesador. El controlador de coherencia responde con una notificación denominada PrRdAck, devolviendo el dato solicitado.

PrWr (*Processor Write*): solicitud de escritura de un dato en el espacio compartido de direcciones por parte del procesador. El controlador de coherencia responde con una notificación denominada PrWrAck.

PrTAS (*Processor Test & Set*): solicitud por parte del procesador de la realización de una operación de tipo *Test & Set* sobre un dato del espacio compartido de direcciones. El controlador de coherencia salva el valor de la variable y modifica su valor a 1 en una única operación atómica, devolviendo el valor salvado a través de una notificación PrTASAck.

PrFAI (*Processor Fetch & Increment*): solicitud por parte del procesador de la realización de una operación de tipo *Fetch & Increment* sobre un dato del espacio compartido de direcciones. El controlador de coherencia incrementa una variable en una única operación atómica, devolviendo el resultado a través de una notificación PrFAIAck.

Cuando el nodo solicita la ejecución de una operación de memoria, el controlador de coherencia la resuelve, devolviendo al procesador una estructura formada por la solicitud realizada, la dirección y el valor del dato al que se ha accedido.

4.4.2 Protocolo entre nodos

Se denomina *evento de protocolo* a cada uno de los mensajes intercambiados a través de la red de interconexión por los controladores de coherencia de los respectivos nodos. Los eventos contienen, además del identificador del tipo de evento, el identificador del bloque al que se hace referencia y los nodos origen y destino del evento. El identificador del bloque recibe el nombre de *descriptor*, y está formado por los bits de etiqueta y de conjunto del bloque al que se hace referencia.

En VSR-COMA existe un total de nueve eventos de protocolo diferentes. Dichos eventos son los siguientes:

BusRreq : solicitud de lectura de un bloque. Va dirigido al propietario del bloque, e incluye el descriptor del bloque y el número del marco donde será almacenado en el nodo local (describiremos la utilidad de este último dato en la sección 4.5).

BusRack : respuesta al evento anterior. Incluye el descriptor del bloque, una copia del mismo y el número del marco donde el bloque está almacenado en el nodo propietario.

BusWreq : solicitud de escritura de un bloque. Va dirigido al propietario del bloque, e incluye el descriptor del bloque y el número del marco donde será almacenado en el nodo local.

BusWack : respuesta al evento anterior. Incluye el descriptor del bloque, el propio bloque y el número del marco donde el bloque estaba almacenado en el nodo propietario (tras enviar este evento, el bloque queda invalidado y el nodo local es el nuevo propietario del bloque).

BusFInv : evento utilizado por el nodo propietario de un bloque en estado SharOwn para invalidar el resto de copias que puedan existir de ese bloque. Incluye el descriptor del bloque y el número del marco en donde está almacenado en el nodo propietario. No requiere respuesta.

BusEXreq : solicitud de desalojo de un bloque. Este evento es generado por el propietario de un bloque, y va dirigido al nodo seleccionado por él como más adecuado para almacenar el bloque. Incluye el descriptor del bloque, el marco que ocupa en el actual propietario, y una copia del bloque.

BusEXack : respuesta positiva al evento anterior. El nodo que la genera pasa a ser el nuevo propietario del bloque. El evento incluye el descriptor del bloque y el número del marco que el bloque ocupará en el nuevo propietario.

BusEXnak : respuesta negativa al evento BusEXreq. El nodo que la genera no acepta el bloque, por lo que el nodo propietario deberá buscar otro candidato. Incluye el descriptor del bloque rechazado.

BusRACE : evento de resolución de condiciones de carrera. Se utiliza cuando un nodo recibe una solicitud respecto de un bloque del que ya no es el propietario, o bien cuando el bloque no puede ser accedido por estar pendiente de otra transacción. Contiene el descriptor del bloque al que hacía referencia la solicitud.

Por otra parte, denominaremos *transacción* al par formado por el evento de solicitud y el de respuesta, independientemente de lo que indique ésta.

4.4.3 Estados de los bloques

Los *estados* en los que puede hallarse un bloque en VSR-COMA pueden dividirse en estados estables y estados transitorios. La posibilidad de que se solapen transacciones en el bus obliga a utilizar estados transitorios, al objeto de indicar que un bloque determinado está pendiente de la finalización de una transacción y no puede ser accedido hasta que ésta acabe.

Los estados en los que puede encontrarse un bloque son los siguientes:

Inv : la información presente en el marco de bloque correspondiente es inválida. La información de etiqueta asociada indica qué bloque estaba almacenado allí antes de que el estado del marco de bloque pasara a ser Inv.

Shared : el nodo puede acceder al contenido del bloque a través de una operación de lectura, pero no de escritura ni de tipo TAS o FAI, ya que el nodo no es el propietario del bloque.

SharOwn : el nodo puede acceder al contenido del bloque a través de una operación de lectura, pero no de escritura ni de tipo TAS o FAI. El nodo es el propietario del bloque, pero posiblemente existan otras copias de ese bloque en otros nodos.

Excl : el nodo puede acceder al contenido del bloque a través de una operación de lectura, escritura, TAS o FAI. No existen otras copias en el sistema. El nodo es el propietario del bloque.

InvWExcl : es un estado transitorio que señala que el contenido del marco de bloque es inválido, pero que el nodo está pendiente de completar una transacción de escritura de un bloque que será almacenado en dicho marco. El nodo no es aún el propietario del bloque. La información de etiqueta indica qué bloque se espera recibir.

InvWShar : es un estado transitorio que indica que el contenido del bloque es inválido, pero que el nodo está pendiente de completar una transacción para poder leer de él. La información de etiqueta indica qué bloque se espera recibir.

WExport : es un estado transitorio que indica que el bloque es *válido*, pero el nodo está pendiente de desalojar el bloque. No pueden atenderse solicitudes de lectura o escritura sobre él, ya que, pese a que el bloque es válido, el nodo seleccionado para el desalojo puede haberlo aceptado, haber invalidado las restantes copias y haber escrito sobre él antes de que el actual propietario haya procesado el evento de respuesta positiva al reemplazo.

4.4.4 Nodos

Los nodos que intervienen en el intercambio de eventos de protocolo reciben diferentes nombres, según la función que estén realizando con respecto a la transacción en curso. Dichos nombres son los siguientes:

Nodo local : se denomina así al nodo que ha enviado un evento de solicitud de lectura (BusRreq), de escritura (BusWreq) o de reemplazo (BusEXreq). Se trata del nodo que inicia la transacción.

Nodo propietario : se denomina así al nodo que posee la propiedad del bloque sobre el que se está realizando la transacción. Cada bloque sólo tiene un propietario en cada momento.

Nodo origen : se denomina así al nodo origen de un evento de protocolo.

Nodo destino : se denomina así al nodo destino de un evento de protocolo.

Nodo remoto : se denomina así a los nodos que no son origen ni destino de un evento de protocolo.

4.5 Mantenimiento de la información de reemplazo

Como se ha descrito en la sección 4.3, el controlador de coherencia mantiene tres estructuras de datos: la información de directorio, la información de estado y etiqueta para la memoria de atracción local, y la información de reemplazo, que contiene la información de estado y etiqueta para cada marco de bloque de todas las memorias de atracción remotas.

Para actualizar la información de reemplazo, el controlador de coherencia utiliza la información contenida en los eventos que recibe a través de la red, ya que dichos eventos permiten deducir lo que sucede en las memorias de atracción remotas. Por ejemplo, si un nodo solicita un bloque para

lectura, todos los controladores de coherencia podrán deducir, al recibir el evento BusRreq, que el nodo que figura como origen de dicho evento ha modificado el estado de uno de los marcos de bloque del conjunto correspondiente, colocándolo en InvWShar. Del mismo modo, cuando el nodo destino responda con un BusRack, todos los controladores de coherencia deducirán que el bloque solicitado estará a partir de este momento en estado SharOwn en el nodo propietario del bloque y en estado Shared en el nodo destino del evento. Lo mismo sucede con el resto de mensajes intercambiados en la red.

Este esquema, sin embargo, presenta un problema. Los eventos intercambiados en la red contienen, además del tipo de evento, el descriptor del bloque en cuestión y los identificadores de origen y destino. Sin embargo, esta información no es suficiente para actualizar la información de estado del bloque en las memorias de atracción remotas, ya que las memorias de atracción son de tipo asociativo por conjuntos. Por lo tanto, el conocimiento del descriptor del bloque permite deducir el conjunto de la memoria de atracción remota relativo a dicho bloque, pero no el marco de bloque que ocupa el bloque en el conjunto, ya que la elección del marco de bloque se realiza examinando asociativamente los contenidos de las etiquetas de dicho conjunto. En consecuencia, un nodo remoto que observa que el nodo origen solicita un bloque determinado puede deducir a qué conjunto se mapeará dicho bloque cuando llegue, pero no a qué marco dentro de dicho conjunto.

Para suplir esa carencia de información, todos los eventos que se generan en VSR-COMA incluyen el número de marco de bloque dentro del conjunto local que se ve afectado por el evento. El número de marco de bloque que se incluye en los eventos es siempre del nodo origen del evento.

El significado del número de marco depende del tipo de evento, como se detalla a continuación:

- En el caso de las solicitudes BusRreq y BusWreq, el número de marco indica el marco que se utilizará para almacenar el bloque y que, por lo tanto, cambia a estado InvWShar e InvWExcl, respectivamente.

- Para las respuestas a los eventos anteriores (BusRack y BusWack), el número de marco indica qué marco ocupa el bloque en el nodo propietario, y que eventualmente cambiará de estado.

- En el caso de la solicitud de desalojo (evento BusEXreq), el número de marco indica el marco de bloque que ocupa el bloque que se pretende desalojar.

- El evento de invalidación rápida (BusFInv) indica el marco de bloque que ocupa el bloque dentro del conjunto correspondiente del nodo propietario.

- Para la respuesta afirmativa de desalojo (evento BusEXack), el número de marco indica en qué marco de bloque se almacena el bloque que ha sido exportado. El evento de respuesta negativa de desalojo (BusEXnak) no lleva identificador de marco de bloque, al no modificarse la situación de los bloques en el nodo que responde negativamente. Lo mismo ocurre con el evento de resolución de condiciones de carrera (BusRACE).

Toda esta información permite mantener actualizado el estado de todos los marcos de bloque de las memorias de atracción remotas. En la sección 4.9 veremos en detalle las acciones que deben realizar los controladores de coherencia al recibir cada uno de los eventos mencionados.

4.6 Premisas de funcionamiento

Todas las transacciones del protocolo entre nodos que se verán a continuación suponen que el procesador local intenta acceder sólo a un bloque. Puede darse el caso de que una operación de memoria haga referencia a un dato que ocupa parte de dos bloques diferentes. Como se verá en la sección 4.11, este caso puede desdoblarse en un par de operaciones sobre dos bloques consecutivos que pueden ejecutarse secuencialmente, devolviendo luego el resultado al procesador.

Al objeto de asegurar la consistencia en el funcionamiento de los controladores de coherencia, se establecen las siguientes premisas de funcionamiento:

- Todo evento generado por un nodo se coloca en el bus común, y es recibido por todos los nodos, *incluyendo él mismo.*
- Existe un orden único de los eventos generados, determinado por el instante en que han sido transmitidos por el bus. Todos los nodos ven los eventos en dicho orden.
- Cada controlador de coherencia procesa los eventos en el orden establecido en el punto anterior.
- Los procesadores sólo generarán una nueva operación de memoria cuando la operación anterior haya sido resuelta por el controlador de coherencia. Por lo tanto, la cola de operaciones de memoria pendientes tendrá un tamaño máximo de 1.
- Los controladores de coherencia de cada nodo sólo atenderán una operación de memoria pendiente cuando no queden eventos pendientes de procesar.

- Una vez que se comienza a atender una operación de memoria, sólo se continúa con la atención de los eventos recibidos a través del bus cuando ocurra alguna de las siguientes situaciones:

  – La operación que está siendo atendida obliga al controlador de coherencia a generar y enviar un evento que requiere una respuesta. En este caso se procesan todos los eventos pendientes hasta que se reciba la respuesta esperada.

  – La operación se completa.

4.7 Funcionamiento general de VSR-COMA

4.7.1 Lectura

Cuando el controlador de coherencia local recibe una operación PrRd por parte del procesador, realiza las siguientes acciones:

1. Si el bloque está presente en algún marco de bloque local (es decir, existe su etiqueta en el conjunto correspondiente y su estado es Shared, SharOwn o Excl), lee el dato y lo entrega al procesador. En este caso se dice que la lectura se ha resuelto de forma *local*.

2. En caso contrario, el controlador de coherencia local deberá solicitar de forma *remota* una copia del bloque a su propietario. Para ello, el nodo local selecciona un marco de bloque en donde almacenar dicha copia. La selección del marco se realiza a través de una política basada en la información de estado [73]. En nuestro caso, la selección se realiza como sigue:

   (a) Si existen bloques en estado Inv en el conjunto, se selecciona aleatoriamente el marco que ocupa uno de ellos.

   (b) En caso contrario, si existen bloques en estado Shared, esto es, copias de bloques de los que el nodo local no es propietario, se selecciona aleatoriamente el marco de bloque que ocupa uno de ellos.

   (c) Si no existen bloques Inv ni Shared significa que el nodo local es propietario de todos los bloques presentes en ese conjunto de la memoria de atracción. Por lo tanto, se deberá desalojar un bloque (trataremos el tema del desalojo en la sección 4.7.3). Como resultado de la transacción de desalojo, el bloque solicitado a través de la operación de memoria de lectura puede haber sido inyectado en la memoria de atracción local. En este caso, la operación de memoria se reinicia desde el paso 1. De no ser así,

la transacción de desalojo genera al menos un marco de bloque libre.

(d) El marco de bloque libre seleccionado para recibir el bloque pasa a estado InvWShar. Su etiqueta cambia a la etiqueta del bloque que va a almacenarse en ese marco.

3. El nodo local busca en la información de directorio la identidad del propietario del bloque que va a solicitar. Una vez hecho esto, el nodo local ya dispone de la información necesaria para solicitar el bloque (figura 4.2).

4. El nodo local envía un evento BusRreq (solicitud de lectura) al nodo propietario, indicando en qué marco de bloque va a almacenar el bloque solicitado cuando lo reciba, que será el marco cuyo estado ha cambiado a InvWShar. Todos los nodos que reciban este evento (excepto el nodo local) actualizan la información de reemplazo correspondiente al nodo local (figura 4.3).

5. El nodo propietario responde con un evento BusRack, que incluye el bloque solicitado y el número de marco de bloque que ese bloque ocupa en el nodo propietario. Si el bloque se encontraba en el nodo propietario en estado Excl, tras esta transacción el bloque pasa al estado SharOwn en el nodo propietario.

   Al recibir el evento BusRack, los controladores de coherencia de cada nodo realizan las siguientes acciones:

   - Todos los nodos que reciban este evento (excepto el nodo propietario) actualizan la información de reemplazo correspondiente al nodo propietario.
   - Todos los nodos que reciban este evento (excepto el nodo local) actualizan la información de reemplazo correspondiente al nodo local, cambiando el estado del marco de bloque que se encontraba en InvWShar a Shared.

   Si el nodo al que se le ha enviado la solicitud ha dejado de ser el propietario o bien posee el bloque en un estado transitorio, dicho nodo responde al nodo local a través de un evento BusRACE. Todos los nodos que reciban este evento lo ignoran, excepto el nodo local, que deberá reintentar la solicitud.

6. Cuando el nodo local reciba el evento BusRack por parte del nodo propietario, se almacena el bloque y se modifica su estado a Shared.

   Tras la recepción y el procesamiento del evento BusRack por parte de todos los controladores de coherencia, la situación queda como puede verse en la figura 4.4.

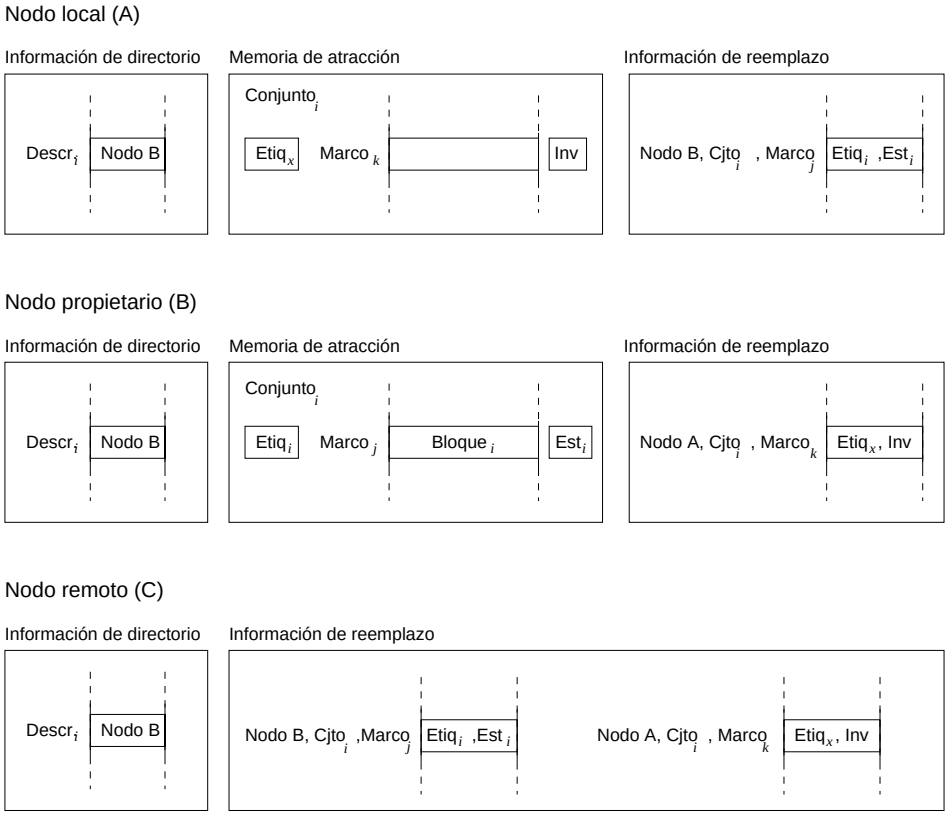


Figura 4.2: Situación inicial en una solicitud de bloque para lectura. El nodo A desea leer un dato presente en el bloque cuyo descriptor es Descr$_i$, compuesto por la etiqueta Etiq$_i$ y el identificador de conjunto Cjto$_i$. El estado de este bloque en el nodo propietario B (Est$_i$) será igual a Excl o SharOwn. El resto de nodo se comportan como nodos remotos. La información de reemplazo indica el estado y la etiqueta de los marcos de bloque involucrados. Sólo se muestra la información relevante para el ejemplo.

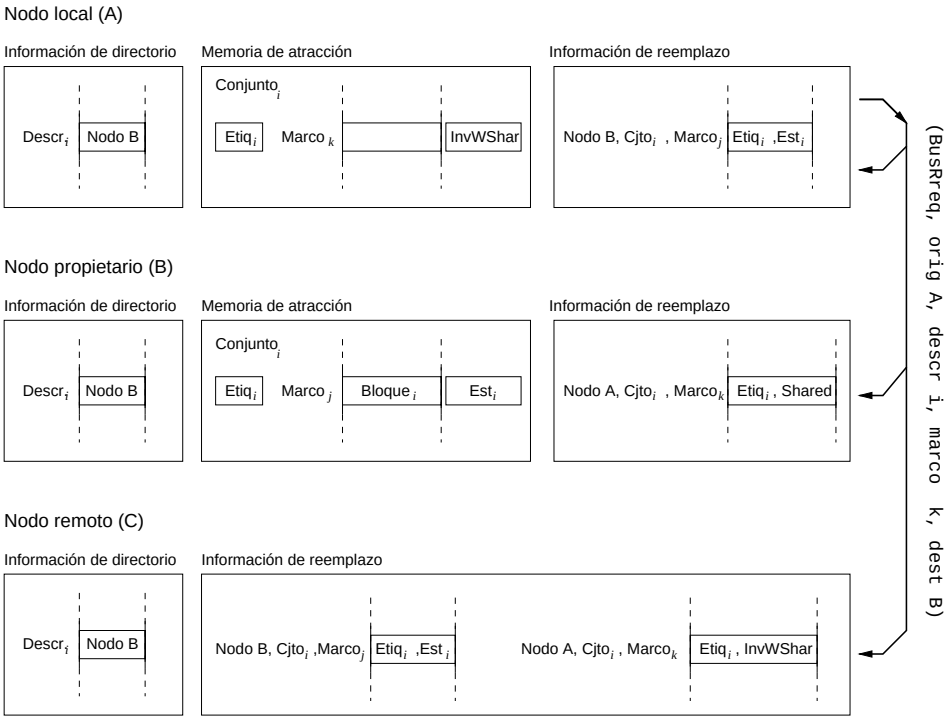


Figura 4.3: Recepción de una solicitud de lectura (evento BusRreq). Cuando el nodo local A envía la solicitud al nodo B, se actualiza la información de reemplazo correspondiente al nodo local en todos los nodos remotos (pasa a InvWShar, a la espera de que se confirme la respuesta del propietario del bloque). En el nodo propietario B, se modifica la información de reemplazo de modo que el estado del marco$_k$ en el conjunto$_i$ del nodo A pase a Shared. Cuando el nodo propietario responda enviando el bloque$_i$, deberá cambiar el estado de su copia (Est$_i$) a SharOwn.

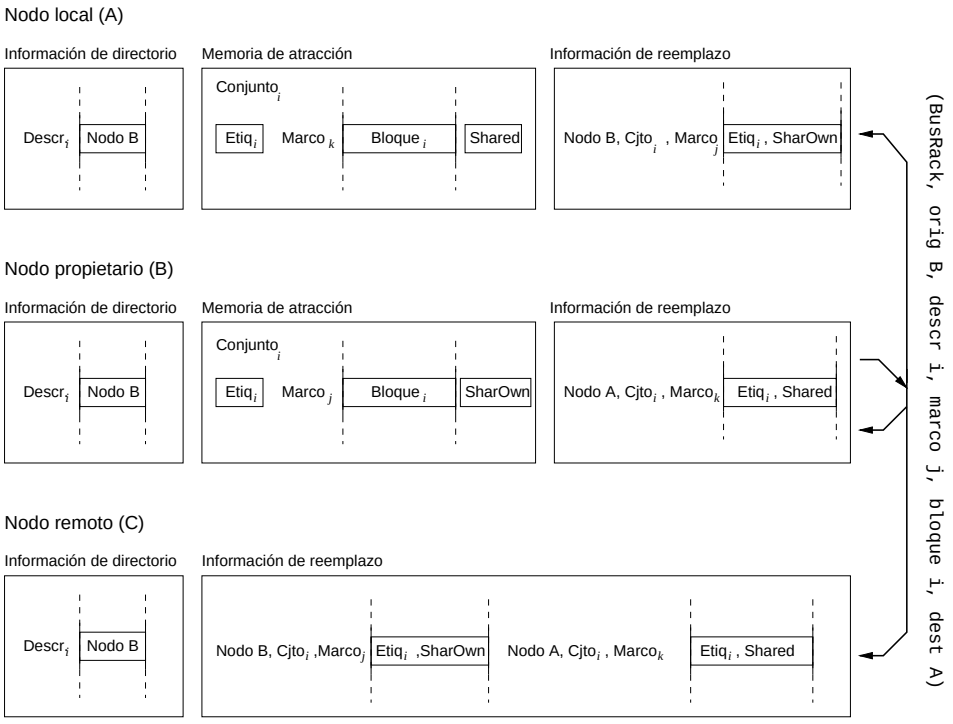


Figura 4.4: Recepción de una respuesta de lectura (evento BusRack). El nodo propietario (B) cambia el estado de su bloque a SharOwn, ya que existe otra copia en el sistema, y envía la respuesta BusRack. Al recibirlo, el nodo local copia el bloque en el marco$_k$ y coloca el estado en Shared. En todos los nodos remotos se actualiza la información de reemplazo correspondientes al nodo local y al nodo propietario. En el nodo local se actualiza la información de reemplazo del nodo propietario. El estado final tras la operación es el que se muestra en la figura.

7. El nodo local devuelve el dato al procesador, completándose la operación de memoria solicitada por éste.

4.7.2 Escritura

Cuando el controlador de coherencia local recibe una operación PrWr, PrTAS o PrFAI por parte del procesador, deberá obtener el bloque en exclusiva. Para ello realiza las siguientes acciones:

1. Si el bloque está presente en algún marco de bloque local y su estado es Excl, el controlador de coherencia local realiza la operación solicitada en dicho bloque, completándose dicha operación de forma local.

2. Si el bloque está presente en algún marco de bloque local y su estado es SharOwn, el nodo local -que además es el nodo propietario del bloque- deberá invalidar el resto de copias que puedan existir en el sistema antes de escribir sobre él. Para ello, primero comprueba en la información de reemplazo si existe alguna otra copia del bloque presente en alguna AM remota. De ser así, se genera un evento BusFInv, indicando el número de marco de bloque que el bloque ocupa en el conjunto correspondiente del nodo propietario. El destino del evento es el mismo que el origen, esto es, el nodo propietario (ver fig. 4.5).

   Al recibir el evento BusFInv, los controladores de coherencia de cada nodo realizan las siguientes acciones:

   - Todos los nodos que reciban este evento, excepto el nodo propietario, actualizan la información de reemplazo correspondiente a la copia maestra del bloque presente en el nodo propietario, pasando su estado de SharOwn a Excl.
   - Todos los nodos que reciban este evento y posean una copia Shared del bloque lo invalidan, pasando su estado a Inv.
   - Todos los nodos que reciban este evento actualizan la información de reemplazo correspondiente a las eventuales copias (en el resto de nodos) del bloque invalidado, buscando todas las copias Shared de ese bloque y cambiando su estado a Inv.

   Una vez hecho esto, el bloque se hallará en estado Excl en el nodo local, pudiéndose completar la operación de memoria. La situación final puede verse en la figura 4.6.

   Si el nodo local posee el bloque en estado SharOwn y comprueba en su información de reemplazo que la única copia existente del bloque es la suya propia, el bloque pasa a Excl sin generarse ningún evento. Esto provocará que la información de reemplazo de los nodos remotos no sea del todo correcta, ya que en éstos figurará el bloque en

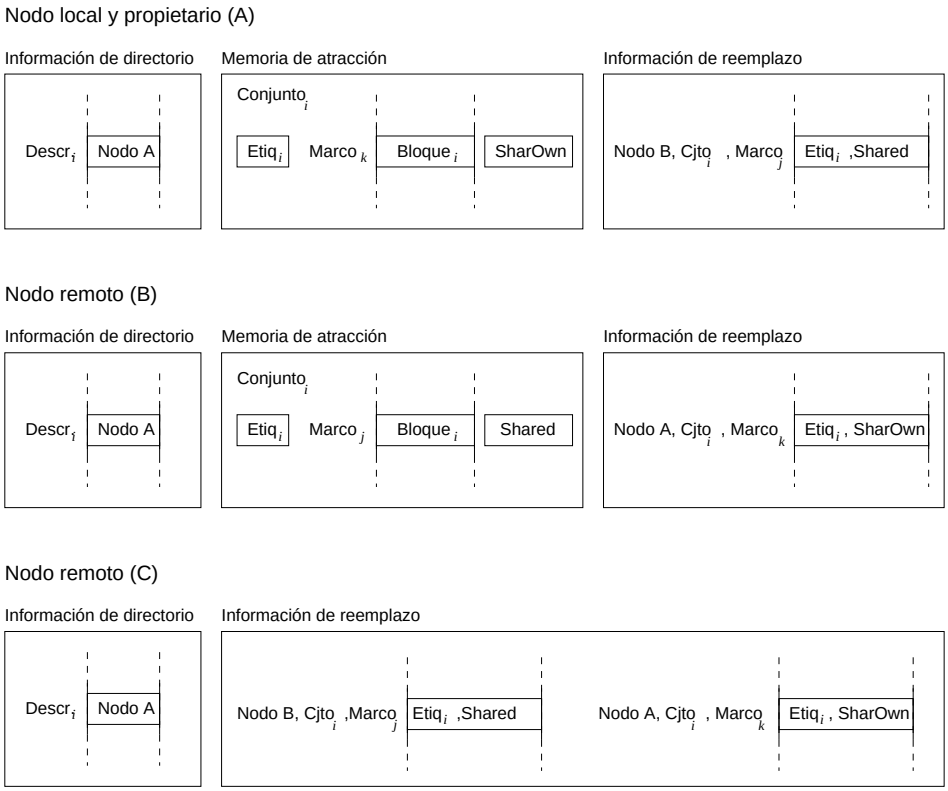


Figura 4.5: Situación inicial en una solicitud de invalidación rápida. El nodo local y propietario (A) posee una copia del bloque en SharOwn, lo que indica que puede haber otras copias válidas del bloque en el sistema. El nodo remoto B posee una de esas copias, mientras que el nodo remoto C no posee ninguna.

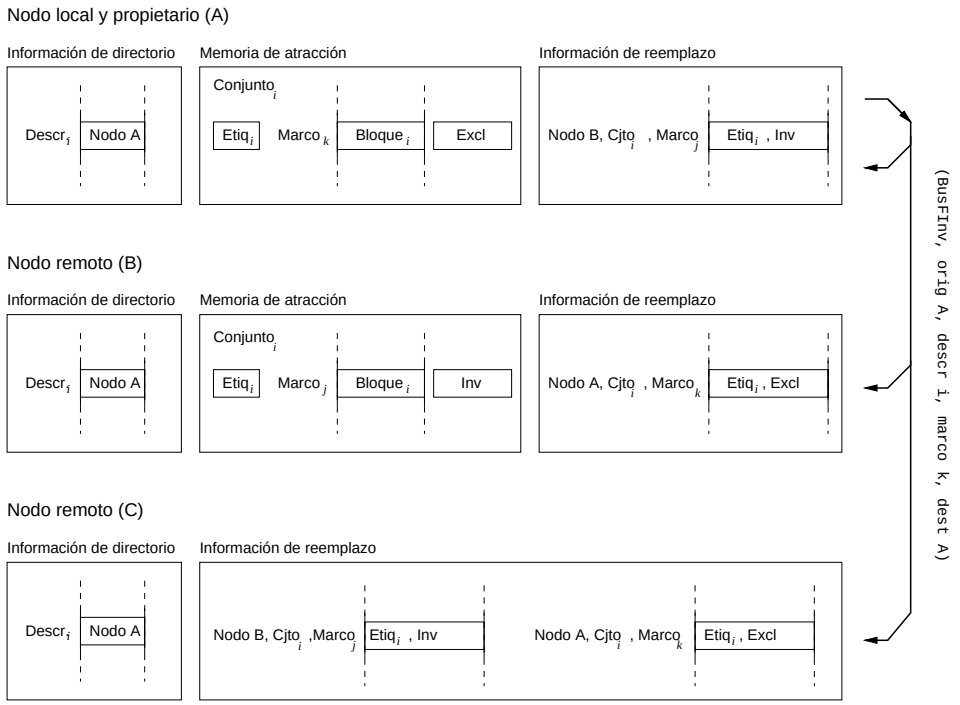


Figura 4.6: Situación final en una solicitud de invalidación rápida. El nodo local y propietario (A) lanza un evento BusFInv. Cuando los nodos remotos lo reciben, actualizan la información de reemplazo correspondiente al nodo local y la correspondiente al resto de nodos que posean una copia. Además, si poseen una copia del bloque, la invalidan. El bloque pasa a Excl en el nodo propietario.

estado SharOwn cuando en realidad está en estado Excl, pero ambos estados reflejan la propiedad del bloque y por lo tanto son equivalentes desde el punto de vista del reemplazo.

3. Si el bloque está presente en algún marco de bloque local y su estado es Shared, no siendo el nodo local el propietario, deberá solicitarse una copia en exclusiva. El estado del bloque pasa a InvWExcl. Su etiqueta se conserva.

4. Si el bloque no está presente en la memoria de atracción local, hay que seleccionar un marco de bloque en donde guardarlo.

   (a) Si existen bloques en estado Inv en el conjunto, se selecciona el marco que ocupa uno de ellos.

   (b) En caso contrario, si existen bloques en estado Shared, esto es, copias de bloques de los que el nodo local no es propietario, se

selecciona el marco de bloque que ocupa uno de ellos.

(c) Si no existen bloques Inv ni Shared significa que el nodo local es propietario de todos los bloques presentes en ese conjunto de la memoria de atracción. Por lo tanto, se deberá desalojar un bloque (trataremos el tema del desalojo en la sección 4.7.3). Como resultado de la transacción de desalojo, el bloque solicitado a través de la operación de memoria de escritura puede haber sido inyectado en la memoria de atracción local. En este caso, la operación de memoria se reinicia desde el paso 1. De no ser así, la transacción de desalojo genera al menos un marco de bloque libre.

(d) El marco de bloque seleccionado para recibir el bloque pasa a estado InvWExcl. Su etiqueta cambia a la etiqueta del bloque que va a almacenarse en ese marco.

5. Una vez seleccionado un marco de bloque en donde almacenar la copia exclusiva de un bloque, el nodo local busca en la información de directorio la identidad del propietario del bloque. Una vez hecho esto, el nodo local ya dispone de la información necesaria para solicitar el bloque (figura 4.7).

6. El nodo local envía un evento BusWreq (solicitud de escritura) al nodo propietario, indicándole en qué marco de bloque va a almacenar la copia exclusiva cuando la reciba, que será el marco cuyo estado ha cambiado a InvWExcl. Todos los nodos que reciban este mensaje -excepto el nodo local- actualizan la información de reemplazo, cambiando a InvWExcl el estado asociado al marco de bloque en donde va a almacenarse el bloque en el nodo local (figura 4.8).

7. El nodo propietario responde enviando el bloque, a través de un evento BusWack (respuesta de escritura), indicando el número de marco de bloque que ocupaba ese bloque en su conjunto correspondiente. El estado del bloque en el nodo propietario pasa a Inv, y se modifica la información de directorio en el nodo propietario de modo que apunte al nuevo nodo propietario -el nodo local-. Además se modifica la información de reemplazo correspondiente al nodo local.

   Al recibir el evento BusWack, los controladores de coherencia de cada nodo realizan las siguientes acciones:

   - Todos los nodos que reciban el evento BusWack y posean una copia Shared de ese bloque la invalidan.
   - Todos los nodos que reciban este evento -excepto el nodo local- actualizan la información de reemplazo correspondiente al nodo

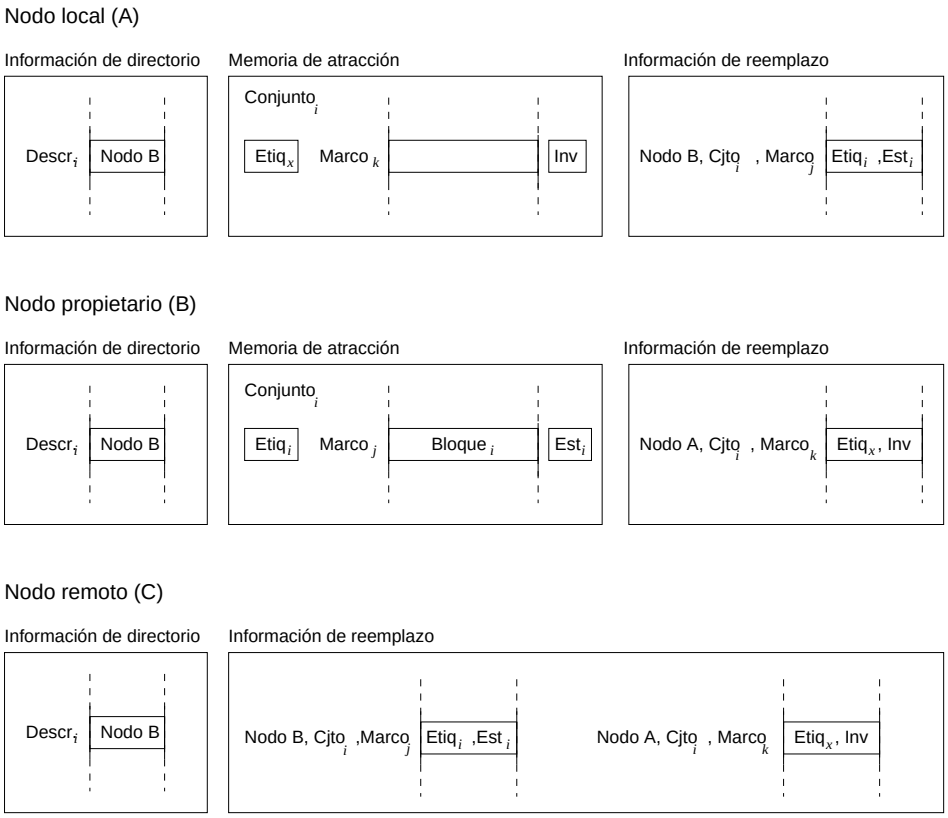


Figura 4.7: Situación inicial en una solicitud de bloque para escritura. El nodo A desea escribir un dato presente en el bloque cuyo descriptor es Descr$_i$, compuesto por la etiqueta Etiq$_i$ y el identificador de conjunto Cjto$_i$. El estado Est$_i$ en el nodo propietario B será igual a Excl o SharOwn. El resto de nodos del sistema se comportan como nodos remotos.

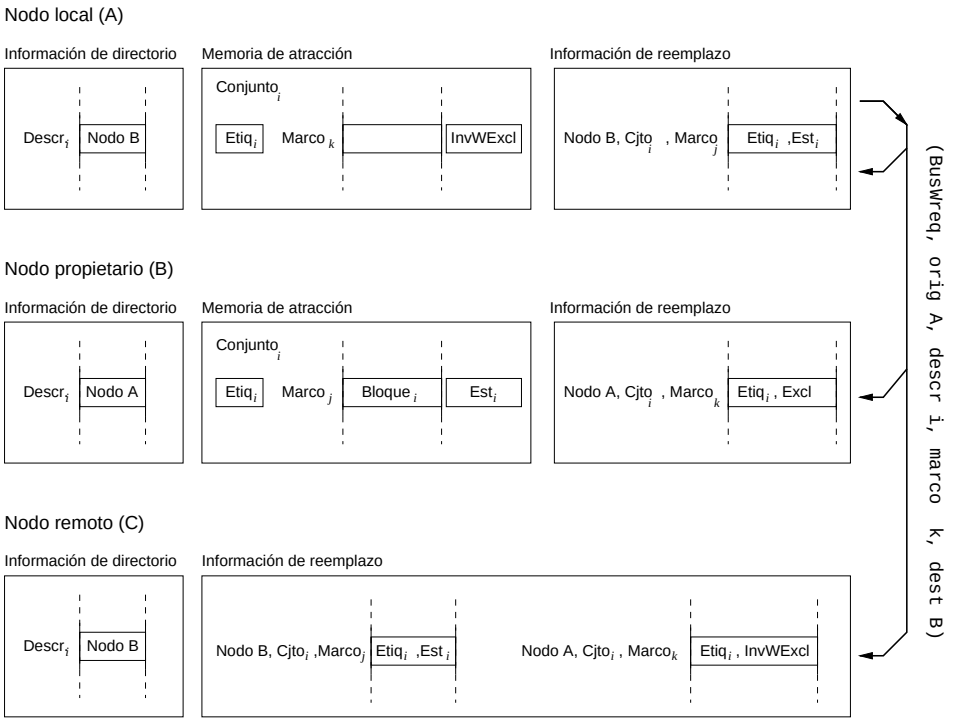


Figura 4.8: Recepción de una solicitud de bloque para escritura (evento BusWReq). Cuando el nodo local (A) envía la solicitud, se actualiza la información de reemplazo correspondiente al nodo local en todos los nodos remotos (pasa a InvWExcl, a la espera de que se confirme la respuesta del propietario del bloque). En el nodo propietario (B) se modifica la información de reemplazo de modo que el estado de Marco$_k$ en el nodo A pase a Excl. También se modifica la información de directorio en el nodo B, pasando a ser el nodo local (A) el nuevo propietario. El bloque se invalidará en el nodo B en cuanto dicho nodo envíe la respuesta BusWack.

local cambiando a Excl el estado asociado al marco de bloque que se encontraba en InvWExcl en el nodo local.

- Todos los nodos que reciban este evento -excepto el nodo que lo ha enviado, esto es, el antiguo propietario- actualizan la información de reemplazo, cambiando a Inv el estado asociado al marco de bloque del nodo origen, y la de directorio, colocando al nodo local como nuevo propietario del bloque.

Si el nodo al que se ha enviado la solicitud de escritura ha dejado de ser el propietario o bien posee el bloque en un estado transitorio, dicho nodo responde al nodo local a través de un evento BusRACE. Todos los nodos que reciban este evento lo ignoran, excepto el nodo local, que deberá reintentar la solicitud.

8. Cuando el nodo local reciba el evento BusWack, almacena el bloque en el marco reservado al efecto y su estado cambia a Excl (figura 4.9).

9. El nodo local realiza la operación solicitada por el procesador, devolviendo su resultado.

4.7.3 Reemplazo

El reemplazo se produce cuando el controlador de coherencia del nodo local necesita un marco de bloque inválido para almacenar un bloque y todos los marcos de bloque del conjunto correspondiente son propiedad del nodo local, bien en estado Excl o SharOwn. El objetivo será entonces transferir a otro nodo la propiedad de uno de esos marcos. Para ello, el controlador de coherencia local realizará las siguientes acciones:

1. El controlador de coherencia del nodo local selecciona uno de sus bloques en estado SharOwn para desalojarlo. Si no hay ningún bloque en SharOwn, se selecciona uno en estado Excl. El motivo de intentar desalojar en primer lugar un bloque en SharOwn es que cabe la posibilidad de que existan copias de ese bloque en otros nodos. En este caso, la operación de desalojo no conllevará la invalidación de ningún bloque en el nodo destino (figura 4.10).

2. El bloque seleccionado pasa a estado WExport.

3. El controlador de coherencia del nodo local selecciona un nodo destino para el desalojo. Como veremos más adelante, la selección del nodo destino es independiente del protocolo. Trataremos este tema en profundidad en la sección 4.10.

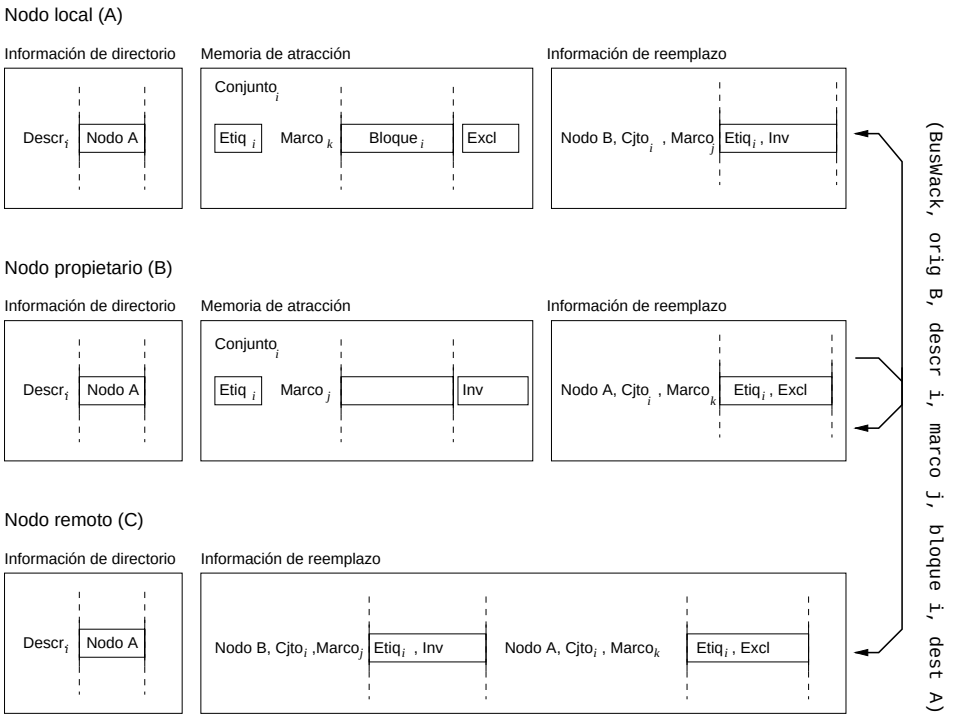


Figura 4.9: Recepción de una respuesta de escritura (evento BusWack). Cuando los nodos reciben este evento actualizan la información de directorio (el nuevo propietario es el A). Todos los nodos actualizan la información de reemplazo. Si algún nodo remoto posee una copia del bloque transferido, la invalida (no es el caso del nodo C). La figura muestra el estado final tras la recepción del evento.

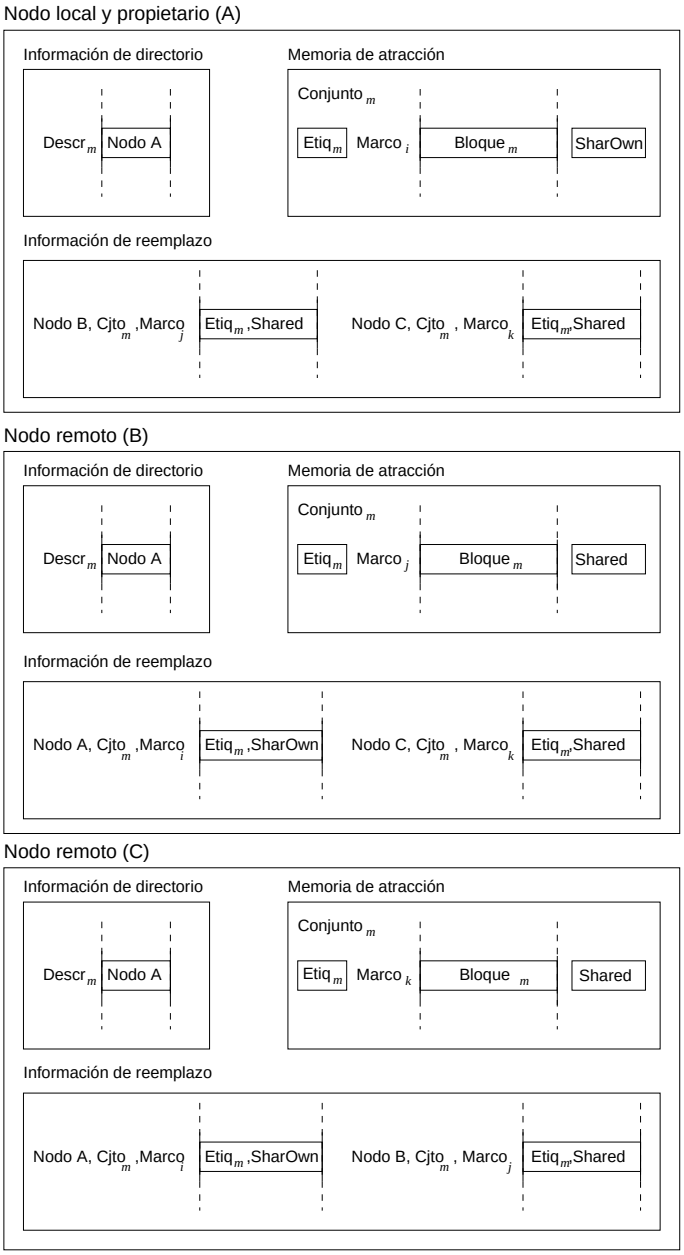


Figura 4.10: Situación previa al desalojo de un bloque. El nodo A desea desalojar el bloque cuyo descriptor es Descr$_m$ y que se encuentra en estado SharOwn. Tiene dos posibilidades: enviarlo al nodo B o al nodo C. Ambos poseen una copia Shared del bloque, por lo que ambas opciones son equivalentes. La selección del nodo destino dependerá del algoritmo de selección correspondiente.

4. El nodo local envía un evento BusEXreq al nodo destino. Este evento incluye el marco de bloque que dicho bloque ocupa actualmente en el nodo local y también el contenido del bloque.

   Todos los nodos remotos que reciban este evento actualizan su información de reemplazo, modificando el estado del bloque que el propietario pretende desalojar a WExport en la información correspondiente al nodo propietario.

5. El nodo destino recibe el evento BusEXreq e intenta almacenar el bloque en el conjunto correspondiente de su memoria de atracción.

   (a) Si dispone de una copia de ese bloque en estado Shared, el nodo destino utiliza ese marco para almacenar el bloque recibido, modificándose su estado a SharOwn.

   (b) En caso contrario, se comprueba si existe un marco de bloque en estado Inv. En ese caso, el nodo destino almacena el bloque recibido en ese marco, cambiando su estado a SharOwn y actualizándose el valor de su etiqueta.

   (c) Si no existen marcos en estado Inv, se comprueba si existe algún bloque en estado Shared en ese conjunto. En ese caso, el nodo destino invalida ese bloque e inmediatamente almacena el bloque recibido en ese marco, cambiando su estado a SharOwn y actualizándose el valor de su etiqueta.

   (d) En caso contrario, significa que el nodo destino de la operación de reemplazo posee todos los bloques del conjunto correspondiente en estado Excl, SharOwn o en estados transitorios. Esto significa que el nodo destino debería desalojar a su vez otro bloque para poder almacenar el bloque recibido. En consecuencia, la solicitud de reemplazo se rechazará.

   Tras la recepción de la solicitud de reemplazo, la situación queda como puede verse en la figura 4.11.

6. Si el nodo destino ha conseguido almacenar el bloque recibido en su memoria de atracción, responde al nodo local con un evento BusEXack, indicando además el número del marco de bloque en donde ha sido almacenado.

   Al recibir el evento BusEXack, los controladores de coherencia de cada nodo realizan las siguientes acciones:

   - Todos los nodos remotos actualizan la información de directorio, pasando el nodo destino a ser el nuevo propietario del bloque desalojado.

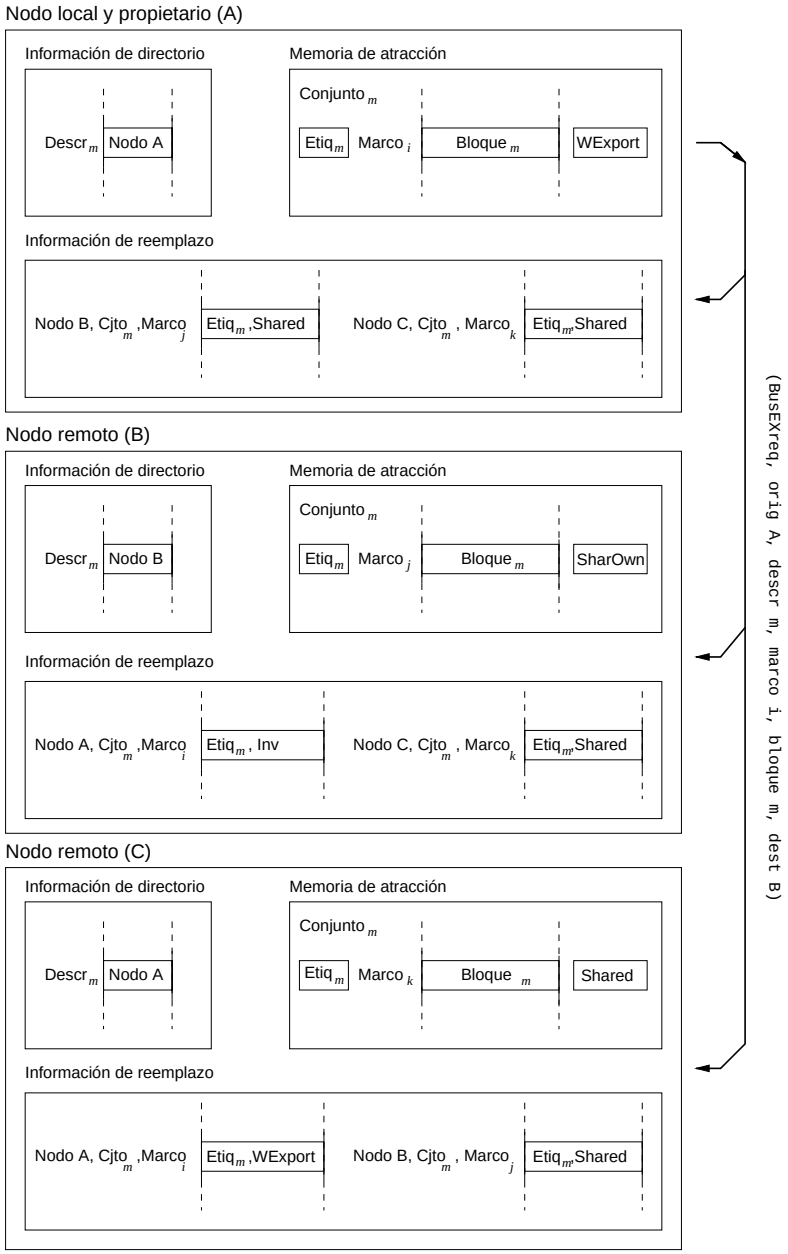


Figura 4.11: Recepción de una solicitud de reemplazo (evento BusEXreq). El nodo A envía la solicitud de reemplazo al nodo B. Al tener una copia del bloque en estado Shared, el nodo B almacena el bloque en ese marco y cambia su estado a SharOwn. El nodo B y el nodo C actualizan la información de reemplazo del nodo local. Nótese que no se invalida la copia presente en el nodo C, ya que el estado del bloque alojado en B es SharOwn, no Excl.

- Todos los nodos remotos actualizan la información de reemplazo correspondiente al nodo local, cambiando el estado de su bloque de WExport a Inv.
- Todos los nodos que reciban el evento, excepto el nuevo propietario -el que ha enviado el evento- actualizan la información de reemplazo correspondiente al nuevo propietario, modificando el estado del marco de bloque en donde ha sido almacenado el bloque desalojado a SharOwn.

7. Cuando el nodo local recibe el evento BusEXack modifica su información de directorio, pasando a ser el nuevo propietario del bloque el nodo que ha enviado el evento. Además modifica el estado del bloque WExport en el nodo local, pasando a ser Inv. Ver figura 4.12.

Si el reemplazo se ha llevado a cabo con éxito, el nodo local dispondrá de un marco de bloque en estado Inv sobre el cual realizar la operación de lectura remota o escritura remota correspondiente.

Cabe sin embargo la posibilidad de que la solicitud de reemplazo deba rechazarse. Consideremos situación mostrada en la figura 4.13. El nodo A solicita una operación de reemplazo al nodo B, pero un instante antes de recibir el evento BusEXreq, el nodo B decide solicitar otro bloque y para guardarlo decide invalidar el mismo marco que quería ocupar el nodo A con su bloque.

En este caso, la operación de desalojo debe rechazarse. Los pasos que se llevan a cabo son los siguientes:

1. Si el nodo destino no puede almacenar el bloque en su memoria de atracción, responde al nodo local con un evento BusEXnak. Todos los nodos remotos actualizan la información de reemplazo correspondiente al nodo local, cambiando el estado de su bloque WExport a SharOwn, ya que el bloque no ha podido ser desalojado.

2. Cuando el nodo local recibe el evento BusEXnak deberá modificar el estado de su bloque WExport, ya que el reemplazo ha sido rechazado y el nodo local continúa siendo el propietario del bloque. El estado del bloque pasa a SharOwn, ya que se ha perdido la información de si el bloque a desalojar era la única copia o existían copias remotas. Ver la figura 4.14.

Al ser rechazada su solicitud de desalojo, el nodo local deberá volver a comprobar si sigue sin existir ningún hueco en su conjunto. De ser así, habrá que reiniciar la operación de desalojo desde el principio.

Cabe aún otra posibilidad. Puede haberse dado el caso de que, en el tiempo que media entre la solicitud de desalojo BusEXreq y su respuesta, tanto positiva como negativa, se haya producido la invalidación de un

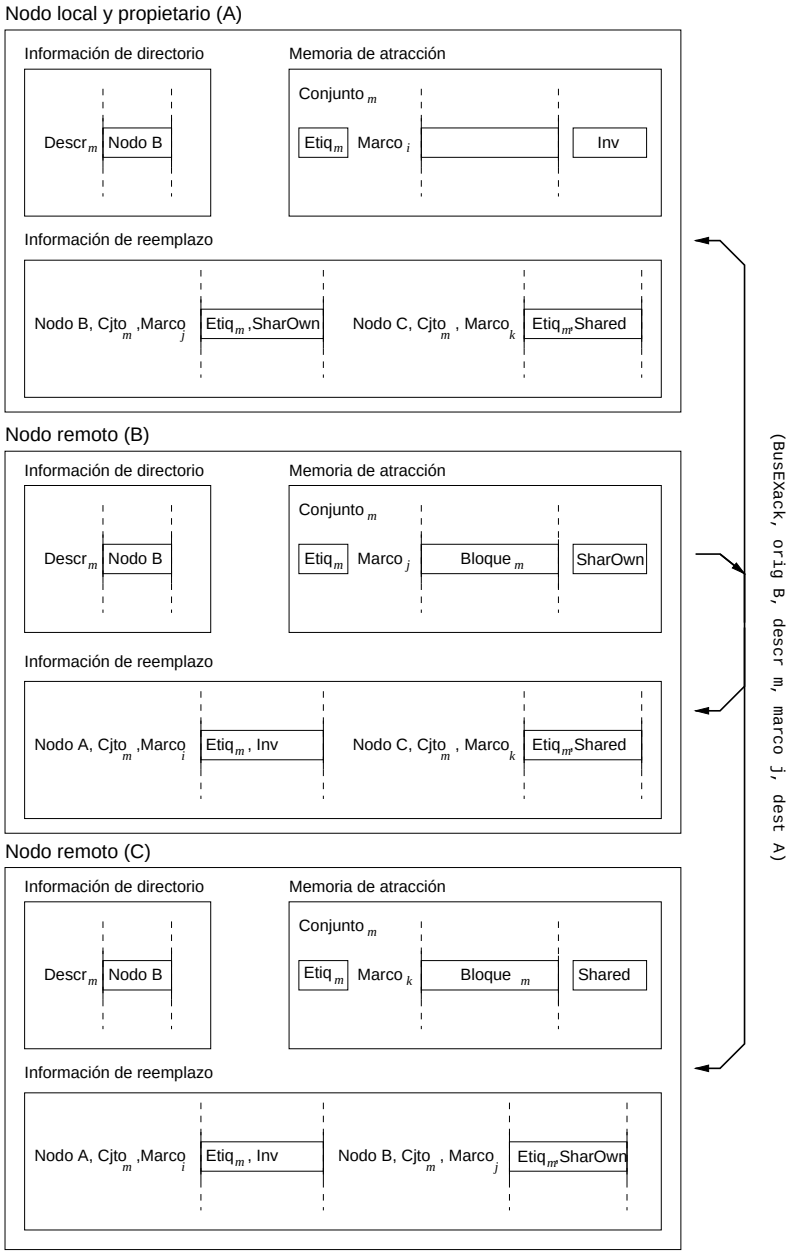


Figura 4.12: Recepción de una respuesta de reemplazo (evento BusEXack). El nodo A y el nodo C actualizan la información de propietario del bloque, junto con la información de reemplazo correspondiente al nodo B. El nodo C, además, actualiza la información de reemplazo del antiguo y del nuevo propietario. El nodo A invalida el bloque. Ya existe un marco de bloque Inv en su memoria de atracción, por lo que puede realizarse la operación de memoria pendiente.

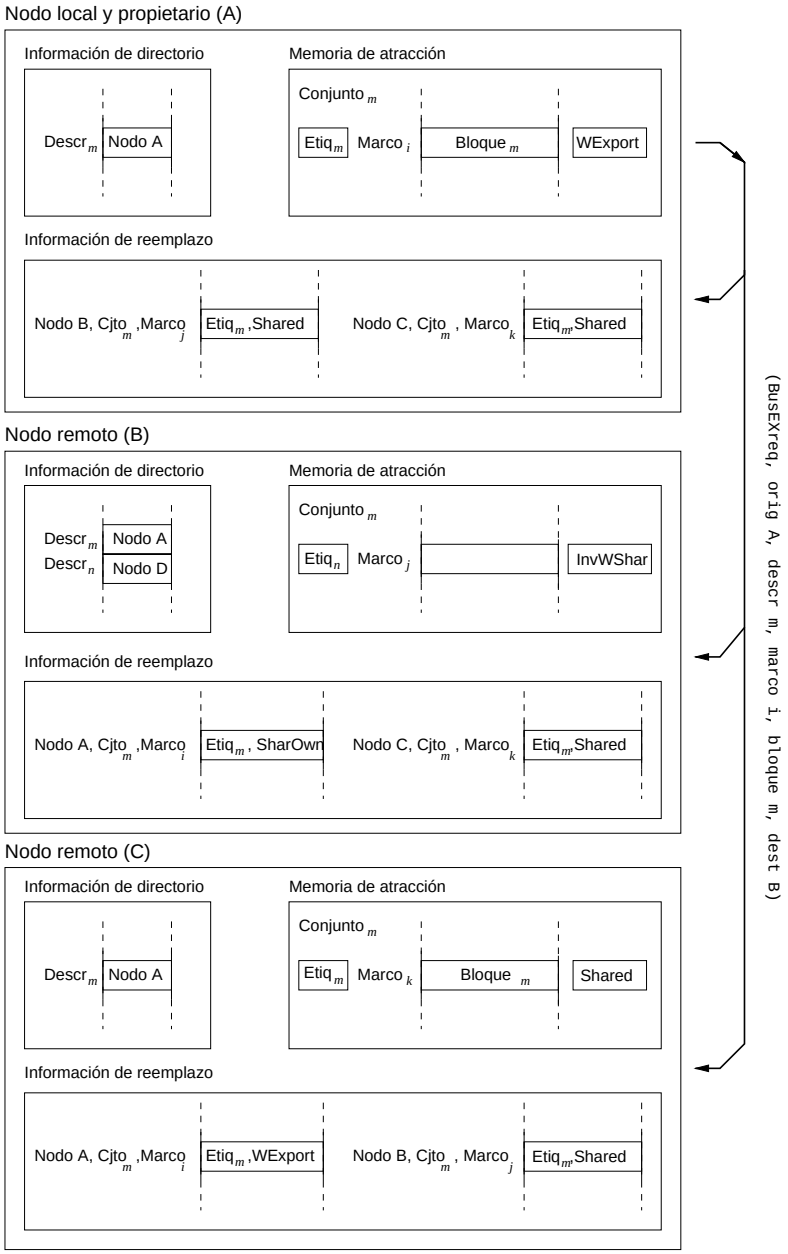


Figura 4.13: Condición de carrera en una solicitud de reemplazo. El estado inicial es el que se muestra en la figura 4.10. Un instante antes de recibir la solicitud de reemplazo del nodo A, el nodo B ha decidido utilizar el marco que ocupa su copia Shared del bloque$_m$ para almacenar una copia del bloque$_n$, que será solicitado de forma remota. En consecuencia, cambia el estado del marco$_j$ a InvWShar. Cuando llegue la solicitud de desalojo, el nodo B deberá responder negativamente (figura 4.14).

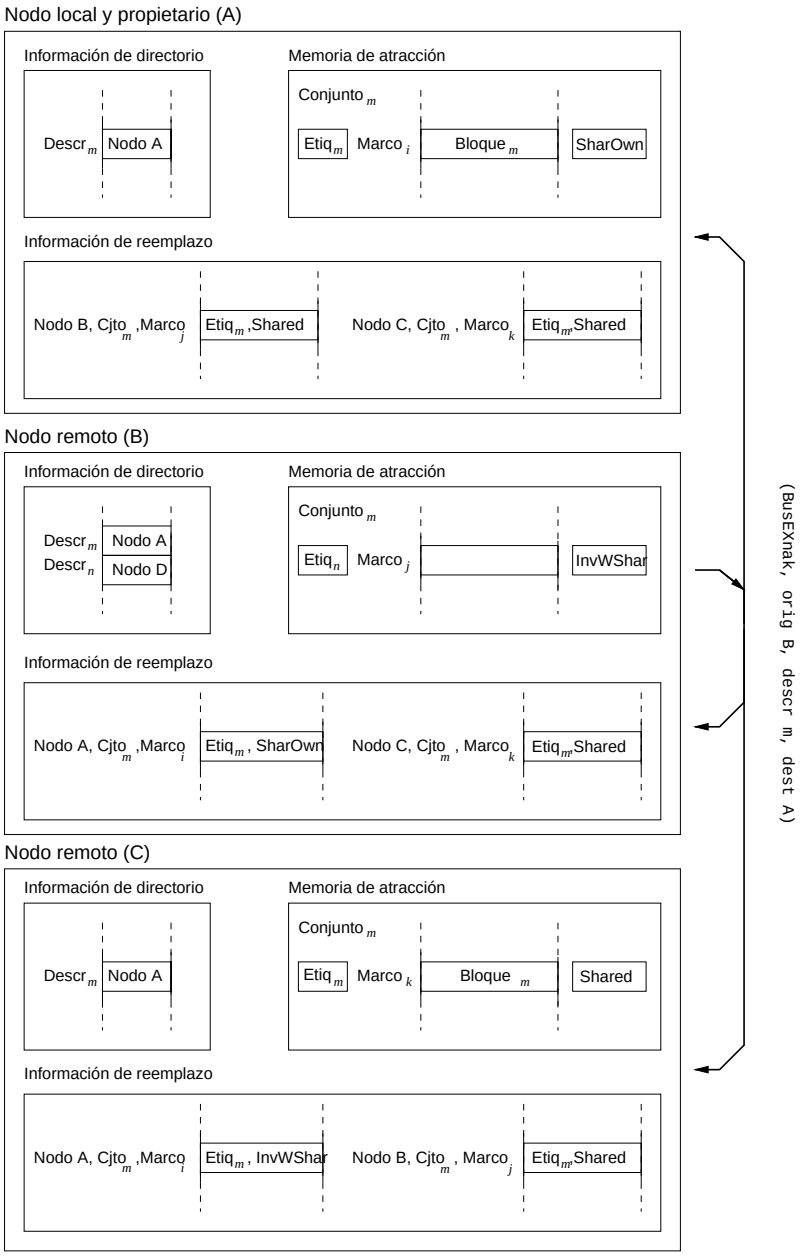


Figura 4.14: Recepción de una respuesta negativa de reemplazo (evento BusEXnak). El nodo propietario del bloque modifica su estado a SharOwn, y el resto de nodos actualizan la información de reemplazo correspondiente a él.

bloque de ese conjunto del cual el nodo local sea propietario. Esto puede suceder si en ese lapso de tiempo un tercer nodo ha solicitado una copia exclusiva de dicho bloque al nodo local. En este caso, cuando se reciba el evento de respuesta al desalojo habrá al menos un bloque Inv en el conjunto, por lo que en el caso de que la respuesta sea negativa no será necesario reintentar el desalojo de un bloque para completar la operación pendiente, al existir un marco libre para almacenar un nuevo bloque.

4.8 Diagrama de estados

La figura 4.15 muestra el diagrama de transiciones de estados del protocolo VSR-COMA para un controlador de coherencia concreto. Todos los controladores de coherencia del protocolo son idénticos, y permiten gestionar la coherencia de forma distribuida. Las transiciones están etiquetadas con el formato siguiente:

Operación Solicitada | Evento Recibido / EventoGenerado | Operación Pendiente | Respuesta Operación

Cada uno de los estados que aparecen en el diagrama se corresponde con uno de los estados posibles de un bloque en la memoria de atracción local. Como puede observarse, se utiliza un estado especial, denominado NotPres, que indica que el bloque no se encuentra en el conjunto correspondiente de la memoria de atracción. Este estado es distinto del estado Inv, ya que en este último caso el contenido del marco de bloque es inválido pero el contenido de la etiqueta se corresponde con la etiqueta del bloque al que se hace referencia. Volveremos sobre el estado NotPres más adelante.

La etiqueta del tipo "Operación Pendiente" que aparece en alguno de los arcos indica que la operación no puede completarse aún. Considérese el arco que va del estado InvWExcl al estado SharOwn. La etiqueta de ese arco indica que la transición se produce cuando se recibe un evento BusEXreq. Esto significa que el propietario del bloque que se ha solicitado a través de un evento BusWreq ha enviado al nodo local el bloque a través de una operación de desalojo. Aunque el nodo local posee el bloque, su propiedad no es exclusiva, por lo que debe cambiar su estado a Excl antes de poder completar la operación de escritura. Las etiquetas del tipo "Operación Pendiente" dan cuenta de este hecho.

Cuando la etiqueta de un bloque no está presente en la memoria de atracción y el controlador de coherencia recibe del procesador una operación de memoria sobre él, las acciones a realizar depende de la situación en la que se encuentre el conjunto correspondiente en la memoria de atracción del nodo local. Pueden darse cuatro situaciones diferentes:

1. Existe al menos un marco de bloque en estado Inv.

Figura 4.15: Diagrama de estados de VSR-COMA.

2. En caso contrario, existe al menos un bloque en estado Shared.

3. En caso contrario, existe al menos un bloque en estado SharOwn.

4. Todos los bloques son Excl.

No puede darse el caso de que exista algún marco de bloque en estado transitorio, ya que el sistema utiliza un modelo de consistencia secuencial y la única operación de memoria en curso es la que se ha iniciado sobre el bloque ausente.

Esta situación y las acciones a realizar en cada una de ellas aparecen reflejadas en la figura 4.16.

Si existe algún marco de bloque en estado Inv o Shared el marco de bloque correspondiente cambia al estado InvWShar o InvWExcl, en función de que la operación de memoria solicitada sea de lectura o escritura. Si todos los bloques del conjunto se encuentran en estado SharOwn o Excl, se hace necesario desalojar un bloque, ya que el nodo es el propietario de todos los bloques presentes en ese conjunto de la memoria de atracción.

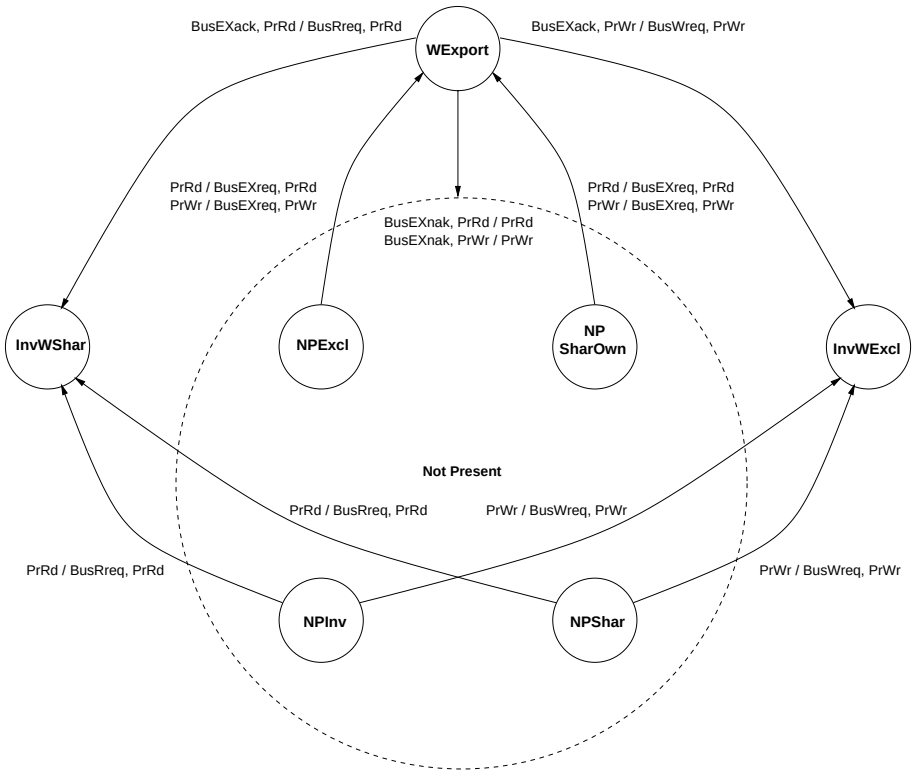


Figura 4.16: Detalle del diagrama de estados de la figura 4.15, para el caso de que el bloque no esté presente en la memoria de atracción.

Por lo tanto, se genera un evento BusEXReq y la operación de memoria solicitada se vuelve a dejar en la cola de operaciones pendientes, a la espera de que se resuelva el desalojo antes de iniciar su procesamiento. Dicha cola tiene una longitud máxima de 1, ya que como se ha dicho el modelo de consistencia utilizado es el modelo secuencial.

Si se recibe como respuesta un evento BusEXack, el bloque ha sido desalojado de la memoria de atracción. En este caso, el marco de bloque que ocupaba el bloque desalojado pasará al estado InvWShar o InvWExcl, según corresponda. Si la solicitud de reemplazo ha sido rechazada, se vuelve a examinar el estado de la memoria de atracción, ya que pudo haberse intercalado una transacción que haya modificado dicho estado.

4.9 Tratamiento de los eventos entrantes

El tratamiento de los eventos de protocolo depende de la identidad del nodo respecto al evento recibido. Como todos los nodos tratan todos los mensajes, pueden darse varios casos: que el nodo que trata el mensaje sea

el que lo ha enviado, que sea el destinatario del mensaje o que sea un nodo remoto. Además, como veremos, el nodo destinatario de un evento puede encontrarse en diferentes situaciones que le permitan responder o no a dicho evento, por lo que examinaremos cada caso en detalle.

Se verá en primer término la nomenclatura que utilizaremos para designar a los nodos y a los eventos entrantes, y a continuación se detallará el comportamiento de cada tipo de nodo para cada uno de los eventos posibles.

4.9.1 Nomenclatura

Se utilizará la nomenclatura **Evento,Nodo [+ Estado]** para designar la llegada de un evento a un tipo de nodo, que eventualmente se encontrará en un cierto estado. El campo **Estado** sólo se tendrá en cuenta cuando el estado del nodo que recibe el evento sea relevante.

Los eventos posibles son los ya descritos en la sección 4.4.2. El formato de cada evento es el siguiente:

- (BusRreq, origen, descriptor, marco_ori, destino)
- (BusRack, origen, descriptor, marco_ori, bloque, destino)
- (BusWreq, origen, descriptor, marco_ori, destino)
- (BusWack, origen, descriptor, marco_ori, bloque, destino)
- (BusFInv, origen, descriptor, marco_ori, destino)
- (BusEXreq, origen, descriptor, marco_ori, bloque, destino)
- (BusEXack, origen, descriptor, marco_ori, destino)
- (BusEXnak, origen, descriptor, destino)
- (BusRACE, origen, descriptor, destino)

Como se ha detallado en la sección 4.4.4, existen diferentes tipos de nodos según su situación respecto a un evento: así, hablamos de nodo local, nodo propietario, nodo origen, nodo destino y nodo remoto.

Cabe destacar una particularidad referida al evento BusFInv. Como se ha descrito más arriba, dicho evento lo genera el nodo propietario de un bloque en estado SharOwn, y tiene como objeto invalidar las eventuales copias Shared del sistema. Para que todos los tipos de eventos tengan un origen y un destino, en el caso del evento BusFInv se ha optado por almacenar el identificador del nodo local en ambos campos.

Se utilizará el término bloque para hacer referencia al bloque, descriptor para hacer referencia al descriptor del bloque (formado por los bits de

etiqueta y los de conjunto), etiqueta para hacer referencia a la etiqueta del bloque, nodo_ori para hacer referencia al nodo origen del evento, etiq_ori para hacer referencia a la etiqueta correspondiente al nodo origen del evento, nodo_des para hacer referencia al nodo destino del evento, marco_ori para hacer referencia al número del marco de bloque en el nodo origen del evento, y marco_des para hacer referencia al número del marco de bloque en el nodo destino del evento.

4.9.2 Solicitud de lectura (evento BusRreq)

Caso 01: BusRreq, Local

Situación: Se produce cuando un nodo N recibe una solicitud de lectura generada por sí mismo y enviada al nodo propietario del bloque.

Acciones a realizar: Ninguna. El evento se descarta.

Estado anterior

- Evento recibido:
  (BusRreq, N, descriptor, marco_ori, nodo_des)
- Conjunto correspondiente de la memoria de atracción:
  marco_N: etiqueta, InvWShar
- Directorio del nodo N:
  descriptor: nodo_des
- Información de reemplazo del conjunto correspondiente: Inexistente para el nodo N.

Estado posterior

- Conjunto correspondiente de la memoria de atracción:
  marco_N: etiqueta,InvWShar
- Directorio del nodo N:
  descriptor: nodo_des
- Información de reemplazo del conjunto correspondiente: Inexistente para el nodo N.
- Evento generado: Ninguno.

Caso 02: BusRreq, Destino + Propietario + bloque válido

Situación: Se produce cuando un nodo N recibe una solicitud de lectura generada por otro nodo. El nodo N es el destino de la solicitud, es el propietario del bloque y el bloque está en estado SharOwn o Excl.

Acciones a realizar: Se actualiza el estado del bloque en la memoria de atracción del nodo N. Se actualiza la información de reemplazo correspondiente al nodo que realiza la solicitud. Se responde a través de un evento BusRack.

Estado anterior

- Evento recibido:
  (BusRreq, nodo_ori, descriptor, marco_ori, N)
- Conjunto correspondiente de la memoria de atracción:
  marco_N: etiqueta, SharOwn | Excl
- Directorio del nodo N:
  descriptor: N
- Información de reemplazo del conjunto correspondiente:
  (nodo_ori, marco_ori): etiq_ori, Shared | Inv

Estado posterior

- Conjunto correspondiente de la memoria de atracción:
  marco_N: etiqueta, SharOwn
- Directorio del nodo N:
  descriptor: N
- Información de reemplazo del conjunto correspondiente:
  (nodo_ori, marco_ori): etiqueta, Shared
- Evento generado:
  (BusRack, N, descriptor, marco_N, bloque, nodo_ori)

Caso 03: BusRreq, Destino + No Propietario

Situación: Se produce cuando un nodo N recibe una solicitud de lectura generada por otro nodo. El nodo N es el destino de la solicitud, pero no es el propietario del bloque.

Acciones a realizar: Se actualiza la información de reemplazo, ya que este evento permite al nodo N saber que el nodo origen desea ese bloque. Se genera un evento BusRACE cuyo destino es el nodo origen.

Estado anterior

- Evento recibido:
  (BusRreq, nodo_ori, descriptor, marco_ori, N)
- Conjunto correspondiente de la memoria de atracción: Irrelevante para este caso.
- Directorio del nodo N:
  descriptor: nodo_distinto_de_N
- Información de reemplazo del conjunto correspondiente:
  (nodo_ori, marco_ori): etiq_ori, Shared | Inv

Estado posterior

- Conjunto correspondiente de la memoria de atracción: Irrelevante para este caso.
- Directorio del nodo N:
  descriptor: nodo_distinto_de_N
- Información de reemplazo del conjunto correspondiente:
  (nodo_ori, marco_ori): etiqueta, InvWShar
- Evento generado:
  (BusRACE, N, descriptor, nodo_ori)

Caso 04: BusRreq, Remoto

Situación: Se produce cuando un nodo N recibe una solicitud de lectura generada por otro nodo. El nodo N no es el nodo destino de la solicitud. El hecho de que sea propietario o no es irrelevante: si lo es, el nodo destino se encargará de devolver un evento BusRACE.

Acciones a realizar: Se actualiza la información de reemplazo, ya que este evento permite al nodo N saber que el nodo origen desea ese bloque.

Estado anterior

- Evento recibido:
  (BusRreq, nodo_ori, descriptor, marco_ori, nodo_des)
- Conjunto correspondiente de la memoria de atracción: Irrelevante para este caso.
- Directorio del nodo N:
  descriptor: nodo_propietario
- Información de reemplazo del conjunto correspondiente:
  (nodo_ori, marco_ori): etiq_ori, Shared | Inv

Estado posterior

- Conjunto correspondiente de la memoria de atracción: Irrelevante para este caso.
- Directorio del nodo N:
  descriptor: nodo_propietario
- Información de reemplazo del conjunto correspondiente:
  (nodo_ori, marco_ori): etiqueta, InvWShar
- Evento generado: Ninguno.

Caso 05: BusRreq, Destino + Propietario + bloque WExport

Situación: Se produce cuando un nodo N recibe una solicitud de lectura generada por otro nodo. El nodo N es el destino de la solicitud y es el propietario del bloque, pero el bloque está en estado WExport.

Acciones a realizar: Se actualiza la información de reemplazo, ya que este evento permite al nodo N saber que el nodo origen desea ese bloque. Se genera un evento BusRACE, indicando que se ha producido una condición de carrera en el tratamiento de ese bloque. Cuando reciba el evento BusRACE, el nodo origen del BusRreq reiniciará la transacción dirigiéndose al nuevo propietario.

Estado anterior

- Evento recibido:
  (BusRreq, nodo_ori, descriptor, marco_ori, N)
- Conjunto correspondiente de la memoria de atracción:
  marco_N: etiqueta, WExport

- Directorio del nodo N:
  descriptor: N
- Información de reemplazo del conjunto correspondiente:
  (nodo_ori, marco_ori): etiq_ori, Shared | Inv

Estado posterior

- Conjunto correspondiente de la memoria de atracción:
  marco_N: etiqueta, WExport
- Directorio del nodo N:
  descriptor: N
- Información de reemplazo del conjunto correspondiente:
  (nodo_ori, marco_ori): etiqueta, InvWShar
- Evento generado:
  (BusRACE, N, descriptor, nodo_ori)

Caso 06: BusRreq, Destino + Propietario + bloque InvWShar | InvWExcl

Situación: Se produce cuando un nodo N recibe una solicitud de lectura generada por otro nodo. El nodo N es el destino de la solicitud y es el propietario del bloque, pero el bloque está en un estado transitorio (InvWShar o InvWExcl).

Acciones a realizar: Se actualiza la información de reemplazo, ya que este evento permite al nodo N saber que el nodo origen desea ese bloque. Se genera un evento BusRACE, para permitir al nodo N completar su transacción. El nodo origen reiniciará la transacción dirigiéndose al nuevo propietario.

Estado anterior

- Evento recibido:
  (BusRreq, nodo_ori, descriptor, marco_ori, N)
- Conjunto correspondiente de la memoria de atracción:
  marco_N: etiqueta, InvWShar | InvWExcl
- Directorio del nodo N:
  descriptor: N
- Información de reemplazo del conjunto correspondiente:
  (nodo_ori, marco_ori): etiq_ori, Inv | Shared | InvWShar

Estado posterior

- Conjunto correspondiente de la memoria de atracción:
  marco_N: etiqueta, InvWShar | InvWExcl
- Directorio del nodo N:
  descriptor: N
- Información de reemplazo del conjunto correspondiente:
  (nodo_ori, marco_ori): etiqueta, InvWShar
- Evento generado:
  (BusRACE, N, descriptor, nodo_ori)

4.9.3 Respuesta de lectura (evento BusRack)

Caso 07: BusRack, Local

Situación: Se produce cuando un nodo N recibe una respuesta de lectura generada por sí mismo y enviada al nodo que inició la transacción de lectura.

Acciones a realizar: Ninguna. El evento se descarta.

Estado anterior

- Evento recibido:
  (BusRack, N, descriptor, marco_ori, bloque, nodo_des)
- Conjunto correspondiente de la memoria de atracción:
  marco_ori: etiqueta, SharOwn
- Directorio del nodo N:
  descriptor: N
- Información de reemplazo del conjunto correspondiente: Inexistente para el nodo N.

Estado posterior

- Conjunto correspondiente de la memoria de atracción:
  marco_ori: etiqueta, SharOwn
- Directorio del nodo N:
  descriptor: N
- Información de reemplazo del conjunto correspondiente: Inexistente para el nodo N.
- Evento generado: Ninguno.

Caso 08: BusRack, Destino

Situación: Se produce cuando un nodo N recibe una respuesta de lectura generada por otro nodo. El nodo N es el destino de la solicitud.

Acciones a realizar: Se almacena el bloque recibido en el conjunto correspondiente de la memoria de atracción local. El marco en el que se almacenará el bloque recibido será aquel que se encuentre en estado InvWShar y cuyo campo de etiqueta coincida con la etiqueta del bloque recibido. El estado del bloque pasa a Shared. Se actualiza la información de reemplazo correspondiente al nodo propietario (el estado del bloque pasa a SharOwn).

Estado anterior

- Evento recibido:
  (BusRack, nodo_ori, descriptor, marco_ori, bloque, N)
- Conjunto correspondiente de la memoria de atracción:
  marco_N: etiqueta, InvWShar
- Directorio del nodo N:
  descriptor: nodo_ori
- Información de reemplazo del conjunto correspondiente:
  (nodo_ori, marco_ori): etiqueta, SharOwn | Excl

Estado posterior

- Conjunto correspondiente de la memoria de atracción:
  marco_N: etiqueta, Shared
- Directorio del nodo N:
  descriptor: nodo_ori
- Información de reemplazo del conjunto correspondiente:
  (nodo_ori, marco_ori): etiqueta, SharOwn
- Evento generado: Ninguno

Caso 09: BusRack, Remoto

Situación: Se produce cuando un nodo N recibe una respuesta de lectura generada por otro nodo, y el nodo N no es el destino de la solicitud.

Acciones a realizar: Se actualiza la información de reemplazo del nodo propietario (el estado del bloque pasa a SharOwn). Se actualiza la información de reemplazo del nodo destino (el estado del bloque que estaba en InvWShar pasa a Shared).

Estado anterior

- Evento recibido:
  (BusRack, nodo_ori, descriptor, marco_ori, bloque, nodo_des)
- Conjunto correspondiente de la memoria de atracción: Irrelevante para el nodo N.
- Directorio del nodo N:
  descriptor: nodo_ori
- Información de reemplazo del conjunto correspondiente:
  (nodo_ori, marco_ori): etiqueta, SharOwn | Excl
  (nodo_des, marco_des): etiqueta, InvWShar

Estado posterior

- Conjunto correspondiente de la memoria de atracción: Irrelevante para el nodo N.
- Directorio del nodo N:
  descriptor: nodo_ori
- Información de reemplazo del conjunto correspondiente:
  (nodo_ori, marco_ori): etiqueta, SharOwn
  (nodo_des, marco_des): etiqueta, Shared
- Evento generado: Ninguno.

4.9.4 Solicitud de escritura (evento BusWreq)

Caso 10: BusWreq, Local

Situación: Se produce cuando un nodo N recibe una solicitud de escritura generada por sí mismo y enviada al nodo propietario del bloque.

Acciones a realizar: Ninguna. El evento se descarta.

Estado anterior

- Evento recibido:
  (BusWreq, N, descriptor, marco_N, nodo_des)
- Conjunto correspondiente de la memoria de atracción:
  marco_N: etiqueta, InvWExcl
- Directorio del nodo N:
  descriptor: nodo_des

- Información de reemplazo del conjunto correspondiente:
  (nodo_des, marco_des): etiqueta, SharOwn | Excl

Estado posterior

- Conjunto correspondiente de la memoria de atracción:
  marco_N: etiqueta, InvWExcl
- Directorio del nodo N:
  descriptor: nodo_des
- Información de reemplazo del conjunto correspondiente:
  (nodo_des, marco_des): etiqueta, SharOwn | Excl
- Evento generado: Ninguno.

Caso 11: BusWreq, Destino + No propietario

Situación: Se produce cuando un nodo N recibe una solicitud de escritura generada por un nodo y dirigida a él. El nodo N no es el propietario del bloque.

Acciones a realizar: Se actualiza la información de reemplazo: el estado del marco de bloque correspondiente en el nodo local pasa a InvWExcl. Se genera una respuesta BusRACE.

Estado anterior

- Evento recibido:
  (BusWreq, nodo_ori, descriptor, marco_ori, N)
- Conjunto correspondiente de la memoria de atracción: Irrelevante para este caso.
- Directorio del nodo N:
  descriptor: nodo_distinto_de_N
- Información de reemplazo del conjunto correspondiente:
  (nodo_ori, marco_ori): etiq_ori, Inv | Shared | InvWExcl

Estado posterior

- Conjunto correspondiente de la memoria de atracción: Irrelevante para este caso.
- Directorio del nodo N:
  descriptor: nodo_distinto_de_N

- Información de reemplazo del conjunto correspondiente:
  (nodo_ori, marco_ori): etiqueta, InvWExcl
- Evento generado:
  (BusRACE, N, descriptor, nodo_ori)

Caso 12: BusWreq, Remoto

Situación: Se produce cuando un nodo N recibe una solicitud de escritura generada por un nodo y enviada al propietario del bloque solicitado. El nodo N no es el origen ni el destino del evento.

Acciones a realizar: Se actualiza la información de reemplazo correspondiente al nodo origen.

Estado anterior

- Evento recibido:
  (BusWreq, nodo_ori, descriptor, marco_ori, nodo_des)
- Conjunto correspondiente de la memoria de atracción: Irrelevante para este caso.
- Directorio del nodo N:
  descriptor: nodo_distinto_de_N
- Información de reemplazo del conjunto correspondiente:
  (nodo_ori, marco_ori): etiq_ori, Inv | Shared | InvWExcl

Estado posterior

- Conjunto correspondiente de la memoria de atracción: Irrelevante para este caso.
- Directorio del nodo N:
  descriptor: nodo_distinto_de_N
- Información de reemplazo del conjunto correspondiente:
  (nodo_ori, marco_ori): etiqueta, InvWExcl
- Evento generado: Ninguno.

Caso 13: BusRreq, Destino + Propietario + bloque WExport

Situación: Se produce cuando un nodo N recibe una solicitud de escritura generada por otro nodo. El nodo N es el destino de la solicitud y es el propietario del bloque, pero el bloque está en estado WExport.

Acciones a realizar: Se actualiza la información de reemplazo, ya que este evento permite al nodo N saber que el nodo origen desea ese bloque en exclusiva. Se genera un evento BusRACE, indicando que se ha producido una condición de carrera en el tratamiento de ese bloque. Cuando reciba el evento BusRACE, el nodo origen del BusWreq reiniciará la transacción dirigiéndose al nuevo propietario.

Estado anterior

- Evento recibido:
  (BusWreq, nodo_ori, descriptor, marco_ori, N)
- Conjunto correspondiente de la memoria de atracción:
  marco_N: etiqueta, WExport
- Directorio del nodo N:
  descriptor: N
- Información de reemplazo del conjunto correspondiente:
  (nodo_ori, marco_ori): etiq_ori, Inv | Shared | InvWExcl

Estado posterior

- Conjunto correspondiente de la memoria de atracción:
  marco_N: etiqueta, WExport
- Directorio del nodo N:
  descriptor: N
- Información de reemplazo del conjunto correspondiente:
  (nodo_ori, marco_ori): etiqueta, InvWExcl
- Evento generado:
  (BusRACE, N, descriptor, nodo_ori)

Caso 14: BusWreq, Destino + Propietario + bloque InvWShar | InvWExcl

Situación: Se produce cuando un nodo N recibe una solicitud de escritura generada por otro nodo. El nodo N es el destino de la solicitud y es el propietario del bloque, pero el bloque está en un estado transitorio (InvWShar o InvWExcl).

Acciones a realizar: Se actualiza la información de reemplazo, ya que este evento permite al nodo N saber que el nodo origen desea ese bloque. Se genera un evento BusRACE, para permitir al nodo N completar su transacción. El nodo origen reiniciará la transacción dirigiéndose al nuevo propietario.

Estado anterior

- Evento recibido:
  (BusWreq, nodo_ori, descriptor, marco_ori, N)
- Conjunto correspondiente de la memoria de atracción:
  marco_N: etiqueta, InvWShar | InvWExcl
- Directorio del nodo N:
  descriptor: N
- Información de reemplazo del conjunto correspondiente:
  (nodo_ori, marco_ori): etiq_ori, Inv | Shared | InvWExcl

Estado posterior

- Conjunto correspondiente de la memoria de atracción:
  marco_N: etiqueta, InvWShar | InvWExcl
- Directorio del nodo N:
  descriptor: N
- Información de reemplazo del conjunto correspondiente:
  (nodo_ori, marco_ori): etiqueta, InvWExcl
- Evento generado:
  (BusRACE, N, descriptor, nodo_ori)

Caso 15: BusWreq, Destino + Propietario + bloque válido

Situación: Se produce cuando un nodo N recibe una solicitud de escritura generada por otro nodo. El nodo N es el destino de la solicitud y es el propietario del bloque. El bloque está en un estado válido (SharOwn o Excl).

Acciones a realizar: Se actualiza la información de reemplazo del que será el nuevo propietario (su estado pasa a Excl). Se actualiza la información de directorio. Se genera y envía un evento de respuesta BusWack. Se invalida la copia existente en la memoria de atracción.

Estado anterior

- Evento recibido:
  (BusWreq, nodo_ori, descriptor, marco_ori, N)
- Conjunto correspondiente de la memoria de atracción:
  marco_N: etiqueta, SharOwn | Excl

- Directorio del nodo N:
  descriptor: N
- Información de reemplazo del conjunto correspondiente:
  (nodo_ori, marco_ori): etiq_ori, Inv | Shared | InvWExcl

Estado posterior

- Conjunto correspondiente de la memoria de atracción:
  marco_N: etiqueta, Inv
- Directorio del nodo N:
  descriptor: nodo_ori
- Información de reemplazo del conjunto correspondiente:
  (nodo_ori, marco_ori): etiqueta, Excl
- Evento generado:
  (BusWack, N, descriptor, marco_N, bloque, nodo_ori)

4.9.5 Respuesta de escritura (evento BusWack)

Caso 16: BusWack, Local

Situación: Se produce cuando un nodo N recibe una respuesta de escritura generada por sí mismo y enviada al nodo que había solicitado el bloque.

Acciones a realizar: Se actualiza la información de reemplazo, ya que todos los nodos que reciban este evento y posean una copia del bloque la invalidarán. El número de actualizaciones realizadas sobre la información de reemplazo dependerá del número de nodos remotos que posean una copia del bloque.

Estado anterior

- Evento recibido:
  (BusWack, N, descriptor, marco_N, bloque, nodo_des)
- Conjunto correspondiente de la memoria de atracción: Irrelevante para este caso.
- Directorio del nodo N:
  descriptor: nodo_des
- Información de reemplazo del conjunto correspondiente:
  (nodo_rem1, marco_rem1): etiqueta, Shared
  (nodo_rem2, marco_rem2): etiqueta, Shared
  . . .

Estado posterior

- Conjunto correspondiente de la memoria de atracción: Irrelevante para este caso.
- Directorio del nodo N:
  descriptor: nodo_des
- Información de reemplazo del conjunto correspondiente:
  (nodo_rem1, marco_rem1): etiqueta, Inv
  (nodo_rem2, marco_rem2): etiqueta, Inv
  . . .
- Evento generado: Ninguno.

Caso 17: BusWack, Destino

Situación: Se produce cuando un nodo N recibe una respuesta de escritura generada por el propietario de un bloque y cuyo destino es el nodo N.

Acciones a realizar: Se almacena el bloque recibido en el conjunto correspondiente de la memoria de atracción local. El marco en el que se almacenará el bloque recibido será aquel que se encuentre en estado InvWExcl y cuyo campo de etiqueta coincida con la etiqueta del bloque recibido. El estado del bloque pasa a Excl. Se actualiza la información de directorio, colocando a N como el nuevo propietario. Se actualiza la información de reemplazo correspondiente al nodo origen del evento (su copia del bloque ha quedado invalidada). Se actualiza la información de reemplazo correspondiente al resto de nodos (sus copias del bloque han sido invalidadas).

Estado anterior

- Evento recibido:
  (BusWack, nodo_ori, descriptor, marco_ori, bloque, N)
- Conjunto correspondiente de la memoria de atracción:
  marco_N: etiqueta, InvWExcl
- Directorio del nodo N:
  descriptor: nodo_ori
- Información de reemplazo del conjunto correspondiente:
  (nodo_ori, marco_ori): etiqueta, SharOwn | Excl
  (nodo_rem1, marco_rem1): etiqueta, Shared
  (nodo_rem2, marco_rem2): etiqueta, Shared
  . . .

Estado posterior

- Conjunto correspondiente de la memoria de atracción:
  marco_N: etiqueta, Excl
- Directorio del nodo N:
  descriptor: N
- Información de reemplazo del conjunto correspondiente:
  (nodo_ori, marco_ori): etiqueta, Inv
  (nodo_rem1, marco_rem1): etiqueta, Inv
  (nodo_rem2, marco_rem2): etiqueta, Inv
  . . .
- Evento generado: Ninguno.

Caso 18: BusWack, Remoto

Situación: Se produce cuando un nodo N recibe una respuesta de escritura generada por el propietario de un bloque y cuyo destino es otro nodo distinto de N.

Acciones a realizar: Si existe una copia en la memoria de atracción local, se invalida. Se actualiza la información de directorio, almacenando la información correspondiente al nuevo propietario. Se actualiza la información de reemplazo correspondiente al nodo origen del evento (su copia del bloque ha quedado invalidada). Se actualiza la información de reemplazo correspondiente al nodo destino del evento (el estado pasa a ser Excl). Se actualiza la información de reemplazo correspondiente al resto de nodos (sus copias del bloque han sido invalidadas).

Estado anterior

- Evento recibido:
  (BusWack, nodo_ori, descriptor, marco_ori, bloque, nodo_des)
- Conjunto correspondiente de la memoria de atracción (suponemos que hay una copia del bloque):
  marco_N: etiqueta, Shared
- Directorio del nodo N:
  descriptor: nodo_ori
- Información de reemplazo del conjunto correspondiente:
  (nodo_ori, marco_ori): etiqueta, SharOwn | Excl
  (nodo_des, marco_des): etiqueta, InvWExcl

(nodo_rem1, marco_rem1): etiqueta, Shared
(nodo_rem2, marco_rem2): etiqueta, Shared
. . .

Estado posterior

- Conjunto correspondiente de la memoria de atracción:
  marco_N: etiqueta, Inv
- Directorio del nodo N:
  descriptor: nodo_des
- Información de reemplazo del conjunto correspondiente:
  (nodo_ori, marco_ori): etiqueta, Inv
  (nodo_des, marco_des): etiqueta, Excl
  (nodo_rem1, marco_rem1): etiqueta, Inv
  (nodo_rem2, marco_rem2): etiqueta, Inv
  . . .
- Evento generado: Ninguno.

4.9.6 Invalidación rápida (evento BusFInv)

Caso 19: BusFInv, Destino

Situación: Se produce cuando un nodo N recibe una invalidación rápida producida y dirigida a sí mismo.

Acciones a realizar: Se actualiza la información de reemplazo, invalidando todas las copias del bloque que se encuentren en estado Shared en el resto de nodos.

Estado anterior

- Evento recibido:
  (BusFInv, N, descriptor, marco_N, N)
- Conjunto correspondiente de la memoria de atracción:
  marco_N: etiqueta, Excl
- Directorio del nodo N:
  descriptor: N
- Información de reemplazo del conjunto correspondiente:
  (nodo_rem1, marco_rem1): etiqueta, Shared
  (nodo_rem2, marco_rem2): etiqueta, Shared

Estado posterior

- Conjunto correspondiente de la memoria de atracción:
  marco_N: etiqueta, Excl
- Directorio del nodo N:
  descriptor: N
- Información de reemplazo del conjunto correspondiente:
  (nodo_rem1, marco_rem1): etiqueta, Inv
  (nodo_rem2, marco_rem2): etiqueta, Inv
- Evento generado: Ninguno.

Caso 20: BusFInv, Remoto

Situación: Se produce cuando un nodo N recibe una invalidación rápida producida por otro nodo.

Acciones a realizar: Se actualiza la información de reemplazo, invalidando todas las copias del bloque que se encuentren en estado Shared en el resto de nodos. Se actualiza la información de reemplazo correspondiente al nodo origen: su estado pasa a Excl. Se actualiza el contenido de la memoria de atracción: si existe una copia del bloque, se invalida.

Estado anterior

- Evento recibido:
  (BusFInv, nodo_ori, descriptor, marco_ori, nodo_ori)
- Conjunto correspondiente de la memoria de atracción (suponemos que hay una copia del bloque):
  marco_N: etiqueta, Shared
- Directorio del nodo N:
  descriptor: nodo_ori
- Información de reemplazo del conjunto correspondiente:
  (nodo_ori, marco_ori): etiqueta, SharOwn
  (nodo_rem1, marco_rem1): etiqueta, Shared
  (nodo_rem2, marco_rem2): etiqueta, Shared

Estado posterior

- Conjunto correspondiente de la memoria de atracción:
  marco_N: etiqueta, Inv

- Directorio del nodo N:
  descriptor: nodo_ori

- Información de reemplazo del conjunto correspondiente:
  (nodo_ori, marco_ori): etiqueta, Excl
  (nodo_rem1, marco_rem1): etiqueta, Inv
  (nodo_rem2, marco_rem2): etiqueta, Inv

- Evento generado: Ninguno.

4.9.7 Solicitud de desalojo (evento BusEXreq)

Caso 21: BusEXreq, Local

Situación: Se produce cuando un nodo N recibe una solicitud de desalojo generada por sí mismo y enviada a otro nodo.

Acciones a realizar: Ninguna. El evento se descarta.

Estado anterior

- Evento recibido:
  (BusEXreq, N, descriptor, marco_N, bloque, nodo_des)

- Conjunto correspondiente de la memoria de atracción:
  marco_N: etiqueta, WExport

- Directorio del nodo N:
  descriptor: N

- Información de reemplazo del conjunto correspondiente: Irrelevante para este caso.

Estado posterior

- Conjunto correspondiente de la memoria de atracción:
  marco_N: etiqueta, WExport

- Directorio del nodo N:
  descriptor: N

- Información de reemplazo del conjunto correspondiente: Irrelevante para este caso.

- Evento generado: Ninguno.

Caso 22: BusEXreq, Remoto

Situación: Se produce cuando un nodo N recibe una solicitud de desalojo generada por el nodo propietario del bloque y enviada a otro nodo. El nodo N no es el origen ni el destino del evento.

Acciones a realizar: Se actualiza la información de reemplazo correspondiente al nodo origen.

Estado anterior

- Evento recibido:
  (BusEXreq, nodo_ori, descriptor, marco_ori, bloque, nodo_des)
- Conjunto correspondiente de la memoria de atracción: Irrelevante para este caso.
- Directorio del nodo N:
  descriptor: nodo_ori)
- Información de reemplazo del conjunto correspondiente:
  (nodo_ori, marco_ori): etiqueta, SharOwn | Excl

Estado posterior

- Conjunto correspondiente de la memoria de atracción: Irrelevante para este caso.
- Directorio del nodo N:
  descriptor: nodo_ori
- Información de reemplazo del conjunto correspondiente:
  (nodo_ori, marco_ori): etiqueta, WExport
- Evento generado: Ninguno.

Caso 23a: BusEXreq, Destino + Espacio libre

Situación: Se produce cuando un nodo N recibe una solicitud de desalojo generada por el nodo propietario del bloque y enviada al nodo N. El nodo N dispone de al menos un marco en estado Inv o bien Shared en donde almacenar el bloque.

Acciones a realizar: Si hay algún marco en estado Inv, se almacena en dicho marco el bloque recibido. En caso contrario, se almacena el bloque en un marco en estado Shared. El estado del bloque almacenado pasará a SharOwn. Se actualiza la información de directorio: el nuevo propietario es el nodo N. Se actualiza la información de reemplazo correspondiente al nodo origen (su copia pasa al estado Inv). Se genera y envía una respuesta BusEXack al nodo origen.

Estado anterior

- Evento recibido:
  (BusEXreq, nodo_ori, descriptor, marco_ori, bloque, N)
- Conjunto correspondiente de la memoria de atracción:
  marco_N: etiq_N, Inv | Shared
- Directorio del nodo N:
  descriptor: nodo_ori
- Información de reemplazo del conjunto correspondiente:
  (nodo_ori, marco_ori): etiqueta, SharOwn | Excl

Estado posterior

- Conjunto correspondiente de la memoria de atracción:
  marco_N: etiqueta, SharOwn
- Directorio del nodo N:
  descriptor: N
- Información de reemplazo del conjunto correspondiente:
  (nodo_ori, marco_ori): etiqueta, Inv
- Evento generado: (BusEXack, N, descriptor, marco_N, nodo_ori)

Caso 23b: BusEXreq, Destino + Espacio ocupado

Situación: Se produce cuando un nodo N recibe una solicitud de desalojo generada por el nodo propietario del bloque y enviada al nodo N. El nodo N no dispone de un marco en estado Inv o bien Shared en donde almacenar el bloque, ya que es el propietario de todos los bloques que ocupan ese marco. Puede darse el caso de que además algún marco esté en estado transitorio (InvWShar, InvWExcl, WExport).

Acciones a realizar: Se genera y envía una respuesta BusEXnak al nodo origen. Cuando dicha respuesta llegue al nodo origen, el estado del bloque que se pretendía exportar pasará de WExport a SharOwn. El nodo N conoce la situación y actualiza la información de reemplazo correspondiente al nodo origen, que pasa a SharOwn.

Estado anterior

- Evento recibido:
  (BusEXreq, nodo_ori, descriptor, marco_ori, N)
- Conjunto correspondiente de la memoria de atracción: Todos los marcos de ese conjunto en estado SharOwn, Excl, InvWShar, InvWExcl, o WExport.
- Directorio del nodo N:
  descriptor: nodo_ori)
- Información de reemplazo del conjunto correspondiente:
  (nodo_ori, marco_ori): etiqueta, SharOwn | Excl

Estado posterior

- Conjunto correspondiente de la memoria de atracción: No se modifica.
- Directorio del nodo N:
  descriptor: nodo_ori
- Información de reemplazo del conjunto correspondiente:
  (nodo_ori, marco_ori): etiqueta, SharOwn
- Evento generado: (BusEXnak, N, descriptor, marco_N, nodo_ori)

4.9.8 Respuesta positiva de desalojo (evento BusEXack)

Caso 24: BusEXack, Local

Situación: Se produce cuando un nodo N recibe una respuesta positiva de desalojo generada por sí mismo y enviada a otro nodo.

Acciones a realizar: Ninguna. El evento se descarta.

Estado anterior

- Evento recibido:
  (BusEXack, N, descriptor, marco_N, nodo_des)
- Conjunto correspondiente de la memoria de atracción:
  marco_N: etiqueta, SharOwn | Excl
- Directorio del nodo N:
  descriptor: N
- Información de reemplazo del conjunto correspondiente: Irrelevante para este caso.

Estado posterior

- Conjunto correspondiente de la memoria de atracción:
  marco_N: etiqueta, SharOwn | Excl
- Directorio del nodo N:
  descriptor: N
- Información de reemplazo del conjunto correspondiente: Irrelevante para este caso.
- Evento generado: Ninguno.

Caso 25: BusEXack, Destino

Situación: Se produce cuando un nodo N recibe una respuesta positiva de desalojo generada por otro nodo y cuyo destino es el nodo N.

Acciones a realizar: Se actualiza la memoria de atracción, invalidando la copia del bloque que se encuentra en estado WExport en el nodo N. Se actualiza la información de directorio, colocando al nodo origen del evento como el nuevo propietario. Se actualiza la información de reemplazo correspondiente al nodo origen del evento (su copia del bloque pasa a SharOwn).

La recepción de este evento finaliza la transacción de reemplazo: el nodo N deberá reiniciar la operación de lectura o escritura que lo llevó a desalojar un bloque.

Estado anterior

- Evento recibido:
  (BusEXack, nodo_ori, descriptor, marco_ori, N)

- Conjunto correspondiente de la memoria de atracción:
  marco_N: etiqueta, WExport
- Directorio del nodo N:
  descriptor: N
- Información de reemplazo del conjunto correspondiente:
  (nodo_ori, marco_ori): etiq_ori, Inv | Shared

Estado posterior

- Conjunto correspondiente de la memoria de atracción:
  marco_N: etiqueta, Inv
- Directorio del nodo N:
  descriptor: nodo_ori
- Información de reemplazo del conjunto correspondiente:
  (nodo_ori, marco_ori): etiqueta, SharOwn
- Evento generado: Ninguno.

Caso 26: BusEXack, Remoto

Situación: Se produce cuando un nodo N recibe una respuesta positiva de desalojo generada por el propietario de un bloque y cuyo destino es otro nodo distinto de N.

Acciones a realizar: Se actualiza la información de directorio, almacenando la información correspondiente al nuevo propietario. Se actualiza la información de reemplazo correspondiente al nodo origen del evento (su copia pasa al estado SharOwn). Se actualiza la información de reemplazo correspondiente al nodo destino del evento (su copia ha quedado invalidada).

Estado anterior

- Evento recibido:
  (BusEXack, nodo_ori, descriptor, marco_ori, nodo_des)
- Conjunto correspondiente de la memoria de atracción: Irrelevante para este caso.
- Directorio del nodo N:
  descriptor: nodo_des

- Información de reemplazo del conjunto correspondiente:
  (nodo_ori, marco_ori): etiq_ori, Inv | Shared
  (nodo_des, marco_des): etiqueta, WExport

Estado posterior

- Conjunto correspondiente de la memoria de atracción: Irrelevante para este caso.
- Directorio del nodo N:
  descriptor: nodo_ori
- Información de reemplazo del conjunto correspondiente:
  (nodo_ori, marco_ori): etiqueta, SharOwn
  (nodo_des, marco_des): etiqueta, Inv
- Evento generado: Ninguno.

4.9.9 Respuesta negativa de desalojo (evento BusEXnak)

Caso 27: BusEXnak, Local

Situación: Se produce cuando un nodo N recibe una respuesta negativa de desalojo generada por sí mismo y enviada a otro nodo.

Acciones a realizar: Ninguna. El evento se descarta.

Estado anterior

- Evento recibido:
  (BusEXnak, N, descriptor, marco_N, nodo_des)
- Conjunto correspondiente de la memoria de atracción: Irrelevante para este caso.
- Directorio del nodo N:
  descriptor: nodo_des
- Información de reemplazo del conjunto correspondiente: Irrelevante para este caso.

Estado posterior

- Conjunto correspondiente de la memoria de atracción: Irrelevante para este caso.

- Directorio del nodo N:
  descriptor: nodo_des
- Información de reemplazo del conjunto correspondiente: Irrelevante para este caso.
- Evento generado: Ninguno.

Caso 28: BusEXnak, Destino

Situación: Se produce cuando un nodo N recibe una respuesta negativa de desalojo generada por otro nodo y cuyo destino es el nodo N.

Acciones a realizar: Se actualiza el estado del bloque en la memoria de atracción, pasando de WExport a SharOwn.

Estado anterior

- Evento recibido:
  (BusEXnak, nodo_ori, descriptor, marco_ori, N)
- Conjunto correspondiente de la memoria de atracción:
  marco_N: etiqueta, WExport
- Directorio del nodo N:
  descriptor: N
- Información de reemplazo del conjunto correspondiente: Irrelevante para este caso.

Estado posterior

- Conjunto correspondiente de la memoria de atracción:
  marco_N: etiqueta, SharOwn
- Directorio del nodo N:
  descriptor: N
- Información de reemplazo del conjunto correspondiente: Irrelevante para este caso.
- Evento generado: Ninguno.

Caso 29: BusEXnak, Remoto

Situación: Se produce cuando un nodo N recibe una respuesta negativa de desalojo generada por el propietario remoto de un bloque y cuyo destino es otro nodo distinto de N.

Acciones a realizar: Se actualiza la información de reemplazo correspondiente al nodo destino del evento (su copia pasa al estado SharOwn).

Estado anterior

- Evento recibido:
  (BusEXnak, nodo_ori, descriptor, marco_ori, nodo_des)
- Conjunto correspondiente de la memoria de atracción: Irrelevante para este caso.
- Directorio del nodo N:
  descriptor: nodo_des
- Información de reemplazo del conjunto correspondiente:
  (nodo_des, marco_des): etiqueta, WExport

Estado posterior

- Conjunto correspondiente de la memoria de atracción: Irrelevante para este caso.
- Directorio del nodo N:
  descriptor: nodo_des
- Información de reemplazo del conjunto correspondiente:
  (nodo_des, marco_des): etiqueta, SharOwn
- Evento generado: Ninguno.

4.9.10 Condiciones de carrera (evento BusRACE)

La recepción de un evento de respuesta BusRACE implica la presencia de una condición de carrera. Como puede observarse en el tratamiento de los eventos entrantes, descritos más arriba, el evento BusRACE sólo se genera en respuesta a una solicitud de lectura o de escritura a un nodo que ya no es el propietario de un bloque.

Por su misma naturaleza, el evento BusRACE puede llegar al nodo destino antes o después de que dicho nodo haya finalizado la transacción. Si llega antes, esto es, si el nodo destino del evento BusRACE recibe dicho evento antes de haber recibido una respuesta válida a su solicitud (sea de lectura o de escritura), el nodo destino deberá reiniciar la operación de memoria que dio origen a la solicitud. Dicha reinicialización implicará volver a comprobar si el bloque está presente en la memoria de atracción local y, si la respuesta es negativa, solicitarlo al nodo propietario, utilizando para ello la información de directorio. Este caso puede darse si el nodo local ha

solicitado un bloque a un nodo que un instante antes ha dejado de ser el propietario del bloque. El nodo destino del evento, que ya no tiene el bloque, genera un BusRACE para indicar que se ha producido una condición de carrera.

Sin embargo, puede suceder que el evento BusRACE llegue a su destino cuando éste ya haya finalizado la transacción correspondiente. Por ejemplo, el nodo local puede haber solicitado un bloque para escritura a través de un evento BusWreq. Mientras está esperando la respuesta, el nodo local recibe un evento BusEXreq que contiene el bloque solicitado. En este caso, el nodo propietario pasará a ser el nodo local, que lo aceptará a través de un evento BusEXack, ya que estaba esperando precisamente ese bloque y tenía espacio preparado para almacenarlo. Con el envío del evento BusEXack, la transacción queda concluida desde el punto de vista del nodo local. Sin embargo, cuando el nodo que ha recibido la solicitud BusWreq responda, lo hará con un evento BusRACE, ya que ha dejado de ser el propietario. Este evento BusRACE, que incluye el descriptor solicitado, llegará a su destino después de que el nodo local haya finalizado su transacción respecto a ese descriptor, y en consecuencia será descartado.

En resumen, si el BusRACE llega a destino antes de que la transacción se complete, dicho evento conlleva el reinicio de la operación de memoria que dio lugar a la transacción. Si llega después, el evento se descarta. Por lo tanto, la llegada de un evento BusRACE nunca da lugar a modificaciones en la información de control (estado y etiqueta, directorio o reemplazo) del controlador de coherencia de ninguno de los nodos, por lo que no recibe ningún tratamiento desde el punto de vista de la gestión de eventos entrantes.

4.10 Selección del nodo destino para desalojo

Una característica importante de VSR-COMA que no aparece reflejada en el diagrama de transición de estados es la selección del nodo destino de una operación de desalojo. Dado que cada nodo dispone de toda la información necesaria para poder elegir el nodo destino, la estrategia de reemplazo puede desligarse del protocolo en sí. Esto posibilita la utilización de diferentes algoritmos de selección de nodo destino sin necesidad de modificar el protocolo VSR-COMA.

A continuación describiremos brevemente la estrategia que utiliza VSR-COMA para la selección del nodo destino. Los bloques SharOwn tienen preferencia para su desalojo sobre los bloques Excl, ya que cabe la posibilidad de que existan copias de ese bloque en otros nodos. El algoritmo es el siguiente:

1. Si existe algún bloque en el conjunto correspondiente que se encuentre en estado SharOwn, se busca un nodo que posea el bloque a de-

salojar en estado Shared. Si existen varios, se selecciona el nodo con menor número de bloques en propiedad en ese conjunto.

2. En caso contrario, se comprueba si algún nodo posee el bloque a desalojar en estado InvWExcl, es decir, está solicitando ese bloque para escritura. Si existen varios, se selecciona el nodo con menor número de bloques en propiedad en ese conjunto.

3. En caso contrario, se comprueba si algún nodo posee el bloque a desalojar en estado InvWShar, es decir, está solicitando ese bloque para lectura. Si existen varios, se selecciona el nodo con menor número de bloques en propiedad en ese conjunto.

4. En caso contrario, se busca un nodo que posea el bloque a desalojar en estado Inv. Si existen varios, se selecciona el nodo con menor número de bloques en propiedad en ese conjunto.

5. En caso contrario, se busca un nodo que posea *cualquier* bloque en estado Inv en ese conjunto. Si existen varios, se selecciona el nodo con menor número de bloques en propiedad en ese conjunto.

6. En caso contrario, se busca un nodo que posea *algún* bloque en estado Shared en ese conjunto. Si existen varios, se selecciona el nodo con menor número de bloques en propiedad en ese conjunto.

7. En caso contrario, se busca un nodo que posea *algún* bloque en estado InvWShar.

8. En caso contrario, se busca un nodo que posea *algún* bloque en estado InvWExcl.

El paso 1 del algoritmo es la solución más lógica si el bloque está en estado SharOwn, ya que existe la posibilidad de que el bloque esté siendo utilizado en otros nodos. De esta forma, se transfiere la propiedad del bloque sin necesidad de que en el nodo destino se invalide un bloque para almacenar el nuevo.

El paso 2 permite desalojar el bloque hacia un nodo que lo esté solicitando para escritura a un propietario erróneo, debido a una condición de carrera. En este caso, en la información de reemplazo consta que el estado de ese marco de bloque es InvWExcl, con la etiqueta igual a la etiqueta del bloque que se desea desalojar. De la misma manera, el paso 3 permite desalojar el bloque hacia un nodo que lo esté solicitando para lectura a un propietario erróneo.

El paso 4 permite desalojar el bloque hacia un nodo que posea un marco en estado Inv con esa etiqueta. Esto indica que el nodo ha utilizado ese bloque en un pasado reciente, lo que es de esperar que disminuya el número

de faltas cache en el futuro, aplicando el principio de localidad temporal en los accesos.

El paso 5 permite desalojar el bloque hacia un nodo que no necesite invalidar ningún bloque para recibirlo, al objeto también de reducir el número de faltas futuras, cuando el nodo intente recuperar una copia del bloque que tuvo que invalidar para recibir al bloque desalojado.

El paso 6 permite enviar el bloque a un nodo con bloques en estado Shared, es decir, copias de las cuales el nodo destino no es propietario, y que por lo tanto pueden eliminarse sin generar tráfico en la red.

Si la presión de memoria del sistema es lo suficientemente alta, puede darse la circunstancia de que, en el instante en el que se intenta el desalojo, todos los marcos de bloque presentes en las memorias de atracción del resto de nodos se encuentren en los estados Excl, SharOwn, InvWShar, InvWExcl o WExport, esto es, estados que indican la propiedad del bloque o bien estados transitorios. El paso 7 del algoritmo intenta desalojar el bloque a un nodo que haya solicitado un bloque para su lectura. En el momento en que el evento BusEXreq llegue a ese nodo pueden darse varias situaciones:

- Que la solicitud de lectura de ese nodo se haya completado, así como la operación de memoria correspondiente. En este caso, habrá en ese conjunto un bloque Shared que podrá sobreescribirse con el bloque enviado.

- Que la solicitud de lectura no se haya completado, pero se haya producido la invalidación de algún bloque de forma remota. Esto habrá generado un marco Inv en donde podrá almacenarse el bloque enviado.

- Que la solicitud de lectura no se haya completado y no se haya invalidado ningún bloque. En este caso, el nodo destino responderá negativamente a la solicitud de desalojo, con lo que el nodo local deberá volver a comprobar el estado de su conjunto y reiniciar, en su caso, el desalojo.

Si no existe ningún marco de bloque remoto en estado InvWShar, el paso 8 del algoritmo intenta desalojar el bloque hacia un nodo que haya solicitado un bloque para escritura, esto es, que posea algún marco de bloque en estado InvWExcl. Las probabilidades de que ese nodo acepte el bloque son bastante bajas: para que esto suceda, otro nodo tiene que haber solicitado un bloque en escritura a ese nodo y la respuesta debe producirse antes de que nuestra solicitud de desalojo llegue a su destino. En ese caso, el nodo destino dispondrá de un marco Inv para recibir el bloque. En caso contrario, la solicitud de reemplazo será rechazada, y el nodo local deberá reiniciar el desalojo examinando la nueva situación.

Dado que en los sistemas COMA la presión de memoria es menor que el 100 % (para permitir la replicación de bloques), se garantiza que en alguna memoria de atracción existe al menos un bloque en un estado distinto de los estados que indican propiedad (Excl, SharOwn o WExport). Por lo tanto, el algoritmo de reemplazo propuesto más arriba contempla todos los casos posibles.

Dado que la información de reemplazo se actualiza a través del procesamiento de los mensajes intercambiados por el bus, puede darse la circunstancia de que esa información no esté actualizada en el momento de tomar la decisión sobre el nodo destino, al existir eventos pendientes de tratamiento. Por lo tanto, cabe la posibilidad de que el bloque sea enviado a un nodo que ya no esté en disposición de aceptarlo, ya que para ello debería desalojar a su vez otro bloque. En este caso, el nodo destino responderá a la solicitud con un evento BusEXnak. El nodo local deberá entonces reiniciar el proceso de desalojo. Es interesante destacar que el bus permite definir una secuencia única de eventos, visible por todos los nodos. Por lo tanto, cuando el nodo propietario reciba la respuesta negativa de reemplazo, ya dispondrá de la información de reemplazo actualizada, al haber tenido que procesar todos los eventos anteriores que han modificado la situación, pudiendo así volver a ejecutar el algoritmo de desalojo.

4.11 Acceso a datos situados en límites de bloques

Dado que la unidad mínima de coherencia es el bloque, y dicho bloque tiene un tamaño fijo, puede darse el caso de que el procesador solicite al controlador de coherencia la lectura o escritura de un dato que se encuentre dividido entre un bloque y el siguiente. La resolución de este problema depende del tipo de operación a realizar sobre el dato. Si se trata de una lectura o una escritura, dicha operación puede hacerse en dos partes, solicitando el primer bloque y completando su tratamiento antes de solicitar el segundo. Si se desea garantizar la corrección de la operación, bastará con colocar la operación de memoria dentro de una zona de exclusión mutua en el programa que ejecute el nodo local.

Sin embargo, la situación es diferente cuando se pretende realizar una operación de *Test & Set* o *Fetch & Increment*. En este caso, si el dato se encuentra dividido entre dos bloques contiguos, se hace necesario que ambos bloques se encuentren *simultáneamente* en la memoria de atracción, al objeto de garantizar la atomicidad en el acceso. Si esta condición no se cumple, otro nodo puede estar modificando una de las mitades del dato, con resultados impredecibles. Este problema no puede solucionarse utilizando primitivas de exclusión mutua en el código de la aplicación que acceda al dato, ya que dichas primitivas se construyen precisamente con estas dos operaciones de memoria. La solución a este problema no es sencilla, ya que

VSR-COMA no define ningún estado que permita "reservar" un bloque en la memoria de atracción mientras se solicita otro para actualizar ambos simultáneamente. Dado que este problema no aparece en las operaciones de lectura o escritura, ya que pueden realizarse por partes, asegurándose su corrección a través de primitivas de sincronización, la solución adoptada ha sido permitir la realización de las operaciones PrTAS y PrFAI exclusivamente sobre datos de un byte. De esta manera se garantiza que el dato se encuentra en un único bloque, independientemente del tamaño del mismo. Por otra parte, dichas operaciones no suelen requerir tamaños mayores que el descrito: un dato accedido a través de una operación *Test & Set* sólo recibe dos valores. El caso del *Fetch & Increment* es diferente, ya que no existe un número máximo de valores que pueda adoptar. Sin embargo, dado que dicho valor suele estar relacionado con el número de nodos presentes en el sistema, se ha considerado razonable limitar dicho valor a 256.

El controlador de coherencia es el encargado de comprobar si el dato sobre el que se pretende realizar una operación de lectura o de escritura está contenido en un único bloque o en dos. En este último caso, pueden darse varias posibilidades, dependiendo de el estado de cada uno de los bloques que contienen las partes del dato. Estas posibilidades son las siguientes:

- Los dos bloques están presentes en la memoria de atracción. En este caso se accede primero a uno y luego al otro. Puede darse el caso de que, mientras se accede al primero, el segundo desaparezca de la memoria de atracción local debido a una invalidación remota, en cuyo caso habrá que solicitarlo para poder completar la operación.

- El primer bloque está ausente de la memoria de atracción y el segundo está presente. En este caso se realiza la operación de lectura o escritura sobre la segunda mitad del dato y una vez completada se comprueba si el primero sigue ausente, ya que pudo haber sido enviado por su propietario a través de una operación de reemplazo. Si el primer bloque no está presente se lo solicita.

- El primer bloque está presente y el segundo está ausente. Al igual que en el caso anterior, se realiza primero la operación sobre la mitad del dato ubicada en el bloque que está presente, comprobándose luego si el segundo continúa ausente y solicitándolo en su caso.

- Los dos bloques ausentes. En este caso se solicita el primer bloque y se realiza la operación sobre la primera mitad del dato. Una vez hecho esto se comprueba si el segundo bloque continúa sin ser local: en ese caso, se lo solicita para completar la operación.

4.12 Conclusiones

El protocolo VSR-COMA, un protocolo COMA de gestión distribuida con reemplazo para sistemas multicomputador unidos por bus común, incorpora todas las características necesarias para la gestión de memoria en un sistema COMA: asociatividad por conjuntos, mecanismos avanzados de reemplazo y operaciones de memoria de tipo *Test & Set*, que facilitan la sincronización entre procesos concurrentes. Se ha examinado en detalle el protocolo de memoria y el protocolo entre nodos de VSR-COMA, prestando especial antención al tratamiento de los eventos entrantes y a la resolución de las eventuales condiciones de carrera.

La separación entre la gestión del desalojo de bloques y la estrategia de reemplazo otorga a VSR-COMA una gran flexibilidad, permitiendo la implementación y evaluación de diferentes estrategias de reemplazo sin necesidad de modificar el protocolo, una posibilidad no contemplada en los protocolos COMA actuales.

En los próximos capítulos se verá el funcionamiento del protocolo VSR-COMA respecto de las otras soluciones propuestas en la bibliografía, a través de la ejecución de cargas de trabajo de uso habitual en la evaluación de rendimiento de sistemas de memoria compartida distribuida. Para ejecutar dichas cargas de trabajo se ha desarrollado un simulador denominado EMUCOMA, cuya descripción se realizará en el siguiente capítulo.

Capítulo 5

Simulación de arquitecturas COMA

5.1 Introducción

Una vez descrito en el capítulo 4 el protocolo VSR-COMA, centraremos nuestra atención en las técnicas de simulación utilizadas para evaluar su funcionamiento. El presente capítulo describe el mecanismo de simulación elegido (la simulación basada en ejecución), así como el modelo de programación de aplicaciones que debe utilizarse para efectuar la simulación. Se describirá también el simulador EMUCOMA, utilizado para simular el funcionamiento de los controladores de coherencia de un sistema VSR-COMA durante la ejecución de aplicaciones paralelas. Finalmente, examinaremos los índices de rendimiento devueltos por EMUCOMA y la utilización de los mismos en la estimación del rendimiento del sistema a través de un modelo analítico.

5.2 La simulación basada en ejecución

Para el desarrollo de EMUCOMA se ha utilizado un mecanismo de simulación basado en ejecución. Este mecanismo de simulación captura las referencias a memoria generadas a través de la ejecución de una aplicación paralela y las resuelve a través de la arquitectura simulada. La resolución de las operaciones de memoria solicitadas por la aplicación permite a ésta continuar con la ejecución del programa.

La simulación basada en ejecución no es el único mecanismo existente para simular el funcionamiento de arquitecturas paralelas. Existen otros mecanismos de simulación posibles, como la simulación basada en traza y el modelado analítico. La simulación basada en traza utiliza una lista de referencias a memoria compartida, generada con anterioridad, como entrada del módulo de simulación de la arquitectura. La lista de referencias va

acompañada de un indicador que determina exactamente el instante en el que dicha referencia se produce. Por lo tanto, una vez generada la traza puede utilizarse para reproducir exactamente los accesos a memoria solicitados por cada uno de los procesos que componen la aplicación paralela. La principal desventaja de la simulación basada en traza es la cantidad de información que debe almacenarse, ya que el número de referencias a memoria compartida provocado por la ejecución de una aplicación paralela de las comúnmente utilizadas para comparar rendimientos genera trazas de varios gigabytes.

El modelado analítico, por su parte, se utiliza más para obtener una visión general de la arquitectura que para estudiar en detalle su comportamiento [79]. Consiste en el desarrollo de un modelo matemático que permita predecir el comportamiento de un sistema. Dado que para ello se utilizan un conjunto de ecuaciones matemáticas, sus resultados son exactos y reproducibles. El principal problema que presentan es la dificultad de modelar sistemas con un gran nivel de detalle, ya que esto incrementa considerablemente la complejidad del modelo.

La utilización de técnicas de simulación basadas en ejecución nos permite obtener índices de funcionamiento que pueden utilizarse para comparar el rendimiento de diferentes arquitecturas en la ejecución de aplicaciones paralelas. Entre sus principales ventajas podemos citar la presencia de un comportamiento no determinista en la ejecución (debido al orden de ejecución de los procesos que decide el planificador del sistema operativo) y la ausencia de ficheros de traza. Este enfoque es el utilizado en el simulador DDM [32], en TANGO [23] y en COMA-BC [84].

Para simular el funcionamiento del protocolo VSR-COMA, se ha desarrollado un sistema de simulación basado en la ejecución simultánea de la aplicación paralela y del simulador de arquitectura EMUCOMA. Se capturan todas las referencias a memoria realizadas por la aplicación: los accesos al espacio compartido de direcciones se convierten en solicitudes al simulador de arquitectura, realizadas a través del protocolo de memoria de VSR-COMA. El simulador las resuelve a través del protocolo de coherencia y devuelve a la aplicación el resultado. Un diseño adecuado del simulador permite una ejecución rápida de las aplicaciones, disminuyendo considerablemente el tiempo de simulación.

La utilización de la simulación basada en ejecución obliga a un tratamiento previo de la aplicación paralela a ejecutar. Dicho tratamiento, realizado de forma automática, consiste en la captura de referencias a memoria y su transformación en solicitudes al simulador de arquitectura. Para poder realizar dicha modificación, deben poder distinguirse en la aplicación las referencias al espacio compartido de las locales. Esto obliga a la utilización de un *modelo de programación* determinado, que permita especificar en la aplicación qué datos forman parte del espacio compartido de direcciones. El modelo de programación que deberá utilizarse en la simulación con

EMUCOMA se basa en la utilización de una estructura que engloba a las variables pertenecientes al espacio compartido de direcciones, y en el uso de un conjunto de macros para su gestión. Las macros utilizadas pertenecen al conjunto Argonne Parmacs [65], de uso muy extendido en el desarrollo de aplicaciones paralelas.

5.3 Modelo de programación

Hemos elegido el lenguaje C [49, 62] y el entorno Unix [48, 96] para el desarrollo y ejecución del simulador EMUCOMA. El lenguaje C, con las extensiones de ejecución paralela propuestas en las macros Parmacs, se revela como el más apropiado para el desarrollo tanto del simulador como de las aplicaciones paralelas que lo utilicen. Dado que la simulación basada en ejecución precisa de una ejecución real de la aplicación paralela, se ha decidido utilizar una plataforma de tipo RISC, concretamente la que suministra Sun Sparc, para la generación e instrumentación del código ejecutable. La principal ventaja del uso de arquitecturas RISC en el proceso de simulación es que la captura de direcciones es muy sencilla, al utilizarse un reducido número de modos de direccionamiento. Por otra parte, los índices obtenidos (por ejemplo el número de instrucciones o de accesos a memoria) no son tan dependientes de la arquitectura ni del conjunto de instrucciones como en las máquinas CISC, facilitando la comparación de los resultados obtenidos.

En las siguientes secciones examinaremos el modelo de programación que deberá utilizarse en las aplicaciones paralelas, así como el uso de las macros Parmacs para la gestión de la memoria compartida y de los mecanismos de sincronización.

5.3.1 Estructura de la memoria compartida

Para la ejecución de la aplicación paralela se ha elegido un modelo de programación basado en variables compartidas. Todas las estructuras de datos pertenecientes a la memoria compartida utilizada por la aplicación deben declararse dentro de una estructura global, definiéndose luego un puntero a dicha estructura. Esta declaración deberá realizarse en la zona de declaración de variables globales, esto es, fuera de toda función de la aplicación. Por ejemplo, si nuestra aplicación utilizara las variables compartidas siguientes, declaradas como

```
float num1;
float num2;
char  car;
int  num3;
struct datos {
```

```
      int dato1;
      char vector[100];
   } dat;
```

su definición a través de una estructura global deberá hacerse como sigue:

```
   struct Memoria {
      float num1;
      float num2;
      char  car;
      int  num3;
      struct datos {
         int dato1;
         char vector[100];
      } dat;
   } *Mem;
```

Esta definición obliga al programador a utilizar el puntero `Mem` para referirse a los datos situados en la memoria compartida, lo que permitirá la localización y captura de las referencias al espacio compartido. Así, todos los accesos a memoria que no vayan acompañados del prefijo `Mem->` serán considerados accesos locales, por lo que no se resolverán a través del simulador de arquitectura.

5.3.2 Las macros Parmacs

Las macros Parmacs [65] proporcionan un nivel de abstracción adecuado para la programación de aplicaciones paralelas. Se definen un conjunto de macros que facilitan la utilización de mecanismos de sincronización, de acceso a memoria compartida y de control de procesos, lo que permite desarrollar aplicaciones para su ejecución paralela sin necesidad de conocer la arquitectura subyacente.

El modelo de programación utilizado para el desarrollo de aplicaciones paralelas y su ejecución mediante el simulador EMUCOMA hace un uso intensivo de las macros Parmacs. Entre las principales ventajas de la utilización de dichas macros podemos citar la posibilidad de implementarlas a través de diferentes mecanismos (macros o funciones) y su extendido uso entre la comunidad científica.

Entre las funciones de las macros Parmacs las principales son facilitar la gestión de la memoria compartida y establecer mecanismos de sincronización. A continuación veremos algunas de las macros más utilizadas.

Macros de entorno y de gestión de memoria

Estas macros permiten establecer el inicio y el fin del programa paralelo, así como gestionar la reserva y liberación de memoria. Así, la macro `MAIN_ENV` permite indicar el inicio del programa paralelo, incorporando eventuales declaraciones para su uso dentro de dicho programa. La macro `MAIN_END` indica el final, debiendo ser la última sentencia en ejecutarse. Estas macros deberán aparecer obligatoriamente en todo programa.

Respecto a la gestión de la memoria compartida, la macro `MAIN-INITENV` permite reservar un espacio de memoria para utilizarlo como memoria compartida. Su formato es el siguiente:

```
MAIN-INITTENV(tam-en-bytes)
```

El tamaño definido a través de esta macro deberá ser el mismo para todos las instancias de la aplicación paralela, y al menos igual al espacio de memoria compartida necesario para su ejecución.

Por su parte, la macro `G_MALLOC` se utiliza para reservar memoria dentro del espacio compartido definido a través de `MAIN-INITENV`. Su formato es

```
Mem = G_MALLOC(tam-en-bytes)
```

Esta macro devuelve un puntero a un bloque de memoria compartida del tamaño indicado. Para liberar la memoria compartida se utiliza la macro `G_FREE`, que recibe como parámetro el puntero a la zona compartida que desea liberarse.

Mecanismos de sincronización

Parmacs define un conjunto de macros que permiten el uso de barreras y semáforos, lo que facilita la sincronización de procesos. Estas macros proporcionan al programador de aplicaciones paralelas una visión de la sincronización y del acceso a regiones críticas que es independiente de la arquitectura subyacente. Los procesos de los que conste la aplicación podrán ser ejecutados en sistemas fuertemente o débilmente acoplados, sin necesidad de efectuar cambios en el código de las aplicaciones.

Semáforos Existen diferentes macros Parmacs que permiten trabajar con semáforos, tanto a nivel individual como a través de la definición de vectores de semáforos. Dichas macros son las siguientes:

`LOCKDEC (semáforo)` : se encarga de crear un semáforo. La creación debe realizarse dentro de la estructura de memoria compartida descrita en la sección 5.3.1.

ALOCKDEC (vect-sem, num) : se encarga de crear un vector de num semáforos.

LOCKINIT (semáforo) : se encarga de inicializar un semáforo. Dado que los semáforos se definen sobre el espacio compartido de direcciones, basta con que un solo proceso realice la inicialización.

ALOCKINIT (vect-sem, num) : se encarga de inicializar un vector de num semáforos.

LOCK (semáforo) : realiza una operación de bloqueo sobre un semáforo.

UNLOCK (semáforo) : realiza una operación de desbloqueo sobre un semáforo.

ALOCK (vect-sem, num) : realiza una operación de bloqueo sobre el semáforo num del vector de semáforos.

AUNLOCK (vect-sem, num) : realiza una operación de desbloqueo sobre el semáforo num del vector de semáforos.

Barreras Las macros Parmacs que permiten la gestión de barreras son las siguientes:

BARDEC (barrera) : se encarga de crear una barrera. Al igual que la creación de semáforos, la creación de barreras debe realizarse dentro de la estructura de memoria compartida.

BARINIT (barrera, valor) : se encarga de inicializar una barrera. Debe ejecutarse una única vez por un solo procesador.

BARRIER (barrera, valor) : se encarga de detener la ejecución del proceso hasta que el contador asociado a la barrera alcance un valor determinado. Cuando se alcanza dicho valor, todos los procesos detenidos se reanudan y la barrera se inicializa automáticamente a cero.

Las macros Parmacs de sincronización pueden ser utilizadas libremente en la aplicación paralela, a condición de que las declaraciones de semáforos y barreras se realicen dentro de la estructura de memoria compartida. De esta manera ambos tipos de primitivas tendrán un ámbito que incluye a todos los procesos de los que consta la aplicación.

5.3.3 Estructura de la aplicación paralela

La estructura elegida para las aplicaciones paralelas es la de procesos independientes. Si la ejecución de una aplicación está formada por n procesos, el modelo de programación utilizado obliga a ejecutar separadamente n instancias de la aplicación, indicando a través de los argumentos pasados a través de la línea de comandos tanto el número total de procesos que forman la ejecución (esto es, el valor de n) como el identificador de proceso correspondiente al proceso actual, que será un valor entre 0 y $n-1$. La función `inicia()`, invocada al principio de la ejecución paralela, se encargará entre otras cosas de procesar los argumentos de entrada y de almacenar en las variables globales `G_NUMNODOS` y `G_NODOLOCAL` el valor de n y el identificador de proceso actual, respectivamente.

Dado que las aplicaciones se ejecutan como procesos independientes, uno de los procesos deberá encargarse de la inicialización de las estructuras de memoria compartida, tanto las propias de la aplicación como las barreras y semáforos que ésta utilice. Suele utilizarse para ello al proceso 0, aunque nada impide elegir a cualquier otro proceso. Por lo tanto, una vez invocada la función `inicia()`, cada proceso deberá comprobar su propia identidad. Si se trata del procesador encargado de las inicializaciones, el proceso deberá llevarlas a cabo. De esta forma se asegura que las tareas de inicialización las realiza un único proceso. Para conseguir que el resto de procesos esperen hasta la finalización de las inicializaciones globales por parte del proceso encargado de esta tarea, suele utilizarse una barrera al final de las mismas.

Una vez visto el modelo de programación, describiremos la estructura que debe tener una aplicación paralela para su ejecución con EMUCOMA. Dicha estructura será la siguiente:

```
// Ficheros de cabecera de la aplicación.
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

// Fichero de cabecera de las rutinas de instrumentación y macros.
#include "../funcie/Emula.h"

// Se invoca a la macro que indica el inicio del codigo paralelo

MAIN_ENV

/////////////////////////////////////////////////////////////////
//  Estructura de memoria compartida.

struct memoria {

  // Datos compartidos de la aplicación de ejemplo.
  char c;
```

```
  int i;
  long int l;
  float f;
  double d;
  int npe;
  int cortes;

  // Barreras de la aplicacion de ejemplo.
  BARDEC( barrera1 )

  // Semaforos de la aplicacion de ejemplo.
  LOCKDEC ( semaforo )
  ALOCKDEC ( semaforo2, 10)

} *Mem;

////////////////////////////////////////////////////////////////
// Programa principal.

main (argc,argv)
int argc;
char **argv;

{

////////////////////////////////////////////////////////////////
// Lo que va a continuacion es obligatorio para toda aplicacion:

// - Se reserva espacio de memoria compartida.

   MAIN_INITENV(1024*1024)

// - Se solicita parte de ese espacio para su uso a través
// del puntero "Mem".

   Mem = (struct memoria *) G_MALLOC ( sizeof(struct memoria) );
   if ( Mem == NULL ) {
     fprintf(stderr, "ERROR: asignacion de memoria compartida.\n" );
     fflush(stderr);
     exit (-1);
   }

//  - Se invoca a la función encargada de procesar los argumentos,
//  de inicializar las comunicaciones del proceso con el simulador
//  de arquitectura y de inicializar las variables compartidas de la
//  simulación.

   inicia (argc, argv);

////////////////////////////////////////////////////////////////

   if (G_nodolocal == 0) {
      // Tareas de inicialización, encargadas aqui al procesador 0.
      // Por ejemplo:
```

```
      BARINIT ( &(Mem->barrera1));
      LOCKINIT ( &(Mem->semaforo));
      ALOCKINIT ( &(Mem->semaforo2),10);
      Mem->c = 0;
   }

   // Inicio del fragmento de código a ejecutar de forma paralela:
   // Se inicia la contabilidad (operación PrStart).
   iniciaT();

   // Aquí va el código paralelo...

   // Final del fragmento de código a ejecutar de forma paralela:
   // Se detiene la contabilidad (operación PrStop).
   acabaT();

   // Aquí va el código que el proceso "maestro" deba ejecutar para
   // finalizar con éxito la aplicación...

   // Final del proceso.
   // Se indica la finalización a Emucoma (operación PrEnd)
   acaba();

   // Se indica el fin del código.
   MAIN_END
}
```

5.3.4 Generación del código ejecutable

Una vez desarrollada la aplicación paralela a través de la utilización de las macros Parmacs y del mecanismo de definición de memoria compartida descrito en las secciones anteriores, deberá procederse a la generación del código ejecutable. Dicho código será exactamente el mismo para cada uno de los procesos de los que consta la aplicación.

La figura 5.1 muestra esquemáticamente el proceso de obtención del código ejecutable. Una vez desarrollado el código en C de la aplicación, se genera el código en ensamblador correspondiente, utilizando para ello un compilador de C estándar. Para la instrumentación de las aplicaciones se utiliza el módulo Pegaxo [40]. Este módulo cumple varias funciones, entre las que cabe destacar las siguientes:

- Expandir las macros Parmacs utilizadas en el código.
- Capturar las instrucciones de acceso a la memoria, reemplazándolas por llamadas a funciones. En tiempo de ejecución, dichas funciones comprobarán si la referencia corresponde al espacio compartido de direcciones: en ese caso, enviarán la solicitud al controlador de coherencia, que se encargará de resolver el acceso y devolver su resultado.

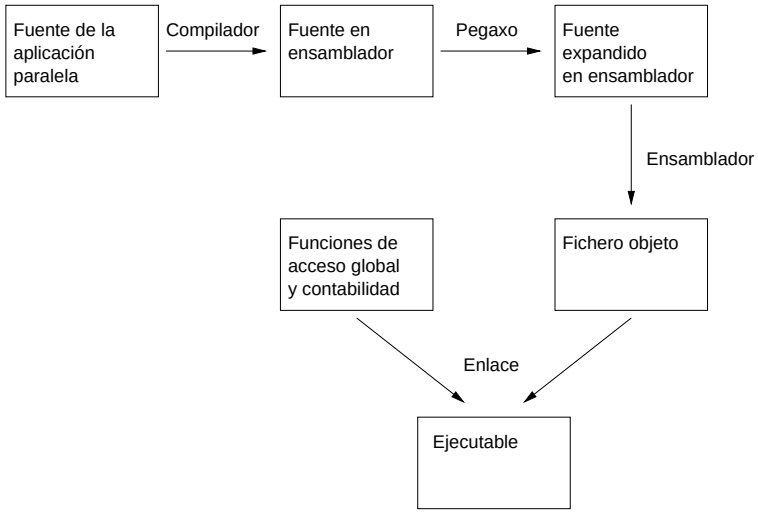


Figura 5.1: Fases de tratamiento del fichero fuente de la aplicación.

- Instrumentalizar el código para llevar la cuenta de las instrucciones ejecutadas por el proceso.

El fichero con el código ensamblador devuelto por Pegaxo se ensambla, obteniéndose un fichero objeto que será enlazado con las funciones que permiten la comunicación con el simulador de arquitectura correspondiente. Una vez realizado el proceso de enlace se obtiene un fichero ejecutable que contiene todo el código necesario para ejecutar la aplicación haciendo uso del simulador de arquitectura EMUCOMA.

5.4 El simulador EMUCOMA

El simulador EMUCOMA es el encargado de suministrar el entorno de ejecución necesario para la simulación de la ejecución de la aplicación paralela utilizando el protocolo de coherencia VSR-COMA. En esta sección veremos los objetivos de diseño del simulador, su arquitectura y su implementación.

5.4.1 Diseño del simulador

El objetivo de diseño básico del simulador EMUCOMA consiste en permitir la ejecución de aplicaciones paralelas convenientemente instrumentalizadas, gestionando el acceso al espacio compartido de direcciones a través de un conjunto de controladores de coherencia que utilicen el protocolo VSR-COMA.

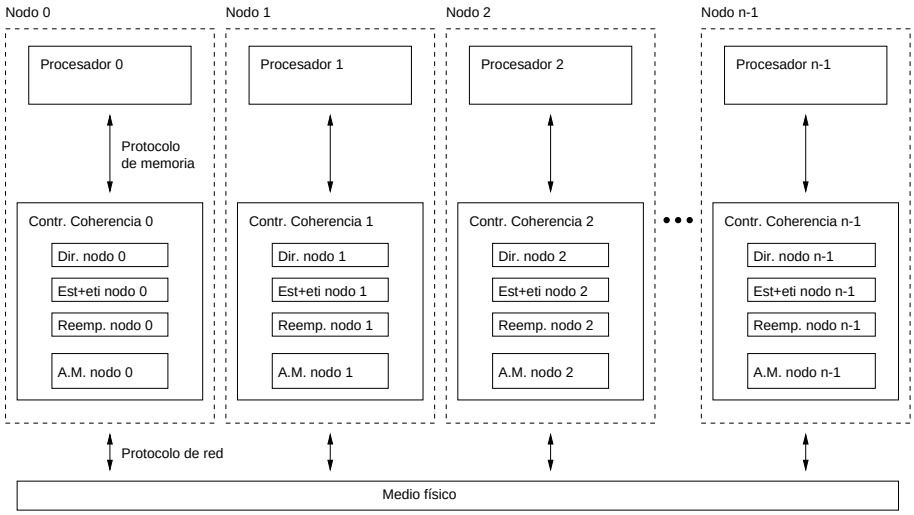


Figura 5.2: Diagrama de bloques del sistema a simular por EMUCOMA.

Como se indicó en la sección 4.10, el mecanismo de selección del nodo destino de una operación de desalojo en el protocolo VSR-COMA es independiente del mismo. Esto permite la utilización de diferentes mecanismos de selección, al objeto de comparar su rendimiento. El diseño modular del simulador EMUCOMA permite modificar fácilmente dicho mecanismo de selección.

El simulador EMUCOMA está escrito completamente en C, funciona en entornos Unix y es independiente de la arquitectura subyacente, a diferencia de las aplicaciones que lo utilizan, que han sido instrumentalizadas a partir del código fuente en ensamblador generado para una arquitectura en concreto. Uno de los objetivos perseguidos con este simulador era permitir una ejecución rápida de las aplicaciones paralelas, utilizando la menor cantidad posible de recursos. El requisito de rapidez en la ejecución ha condicionado el desarrollo de EMUCOMA.

La figura 5.2 muestra el diagrama de bloques del sistema a simular, compuesto de un conjunto de n nodos, numerados del 0 al $n-1$, en cuyos procesadores se ejecuta una aplicación paralela. Todos los nodos se encuentran conectados a través de una red de bus común. Cada nodo está formado por un procesador que ejecuta el programa y un controlador de coherencia que se encarga del mantenimiento de la coherencia del espacio compartido de direcciones.

Los procesadores realizan los accesos a la memoria compartida a través de las operaciones de memoria del protocolo VSR-COMA, ya descritas en la sección 4.4.1. Los controladores de coherencia se encargan de resolver dichos accesos, bien de forma local a través de los valores almacenados en

su memoria de atracción o bien de forma remota. En este último caso, los diferentes controladores utilizarán el protocolo VSR-COMA para obtener los bloques correspondientes. Como veremos, la simulación de la red de bus común se realiza a través de una estructura de datos común a todos los controladores y un conjunto de operaciones asociadas a la misma, lo que permite a cada controlador enviar y recibir los eventos definidos en el protocolo.

5.4.2 Funciones de captura de referencias a memoria compartida

Como se ha dicho en la sección 5.3.4, uno de los principales cometidos del módulo Pegaxo es la captura de las referencias a la memoria compartida. Cada vez que aparece una instrucción de acceso a memoria, Pegaxo la sustituye por una llamada a una función que se encarga de solicitar el dato al controlador de coherencia. Las dos funciones utilizadas para ello (cuyos nombres son `_CAPTURA_read()` y `_CAPTURA_write()`) reciben como parámetros de entrada la dirección del operando fuente, la dirección del operando destino y el tamaño de la operación en bytes. Estas funciones comprueban si la lectura solicitada corresponde a la memoria compartida: de ser así, se solicita al controlador de coherencia que complete la operación. Cuando el controlador de coherencia responde indicando la finalización de la operación, dicha operación se completa de forma local, copiando el dato al destino correspondiente y devolviendo el control al programa principal.

Además de resolver las referencias a memoria compartida, las funciones descritas realizan también operaciones de contabilidad, manteniendo actualizados el número de lecturas y escrituras totales y el número de lecturas y escrituras al espacio compartido.

5.4.3 Funciones de sincronización

Dado que el protocolo VSR-COMA define operaciones de memoria que facilitan la implementación de mecanismos de sincronización (a través de las operaciones de memoria compartida de tipo *Test & Set* y *Fetch & Increment*), se ha optado por expandir las macros como funciones que utilicen dichos mecanismos para sincronizar el acceso de los procesos a las regiones críticas [30]. A continuación veremos cómo se han implementado dichas funciones.

Semáforos

La implementación de semáforos se ha realizado utilizando la operación de memoria PrTAS del protocolo VSR-COMA. Esta operación almacena el valor anterior de la variable y lo cambia a 1 en una única operación atómica. La utilización de esta operación de memoria para la implementación de la

macro LOCK es directa: basta con situar la operación de memoria dentro de un bucle, dando la operación por finalizada cuando el resultado devuelto por la misma sea 0.

Sin embargo, este enfoque para la implementación de semáforos tiene algunos inconvenientes [20]. El principal atañe al hecho de que la operación PrTAS obliga a realizar una escritura remota para comprobar si el valor previo de la variable ha cambiado a 0. Esto lleva a numerosas invalidaciones remotas, ya que para escribir el bloque el controlador de coherencia debe adquirirlo en exclusiva. Si existen varios procesos pendientes de acceder a una sección crítica a través de este mecanismo, esta situación eleva sensiblemente el tráfico de red.

La solución adoptada en la implementación de la macro LOCK es la utilización de un algoritmo denominado *Test & TAS* [20]. Este algoritmo es el siguiente:

```
do {
   do
      PrRd(semaf);
   while (semaf == 1);
   PrTAS(semaf);
}
while (semaf==1);
```

El valor del semáforo se lee dentro de un bucle, que se repetirá mientras el valor sea igual a 1. Cuando el valor cambia a 0, indicando que otro proceso puede acceder al semáforo, se realiza sobre él una operación PrTAS. Si se tiene éxito, el procesador se encuentra en posesión del recurso. En caso contrario, se vuelve a iniciar el bucle de lectura.

Este algoritmo presenta una importante ventaja sobre la utilización del *Test & Set*: el tiempo de espera para que el semáforo se libere se consume en lecturas locales, en lugar de escrituras que generan eventos. En la primera iteración del bucle de lectura, si el bloque que contiene al dato no se encuentra en la memoria de atracción local, se solicita una copia para lectura. A partir de ese momento, todas las lecturas realizadas son locales. Cuando el procesador que se encuentra en la zona de exclusión mutua libera el semáforo, el bloque local será invalidado de forma remota, generándose una falta en la siguiente lectura. En ese momento se producen varios eventos de solicitud de escritura por parte de todos los procesadores que intentan acceder a la sección crítica, hasta que uno lo consigue y el resto queda nuevamente en espera. El número de transacciones generadas para intentar acceder a la sección crítica es $O(N^2)$, siendo N el número de procesadores [66]. Sin embargo, dicho número es mucho menor que si sólo se utilizara el mecanismo *Test & Set*.

La implementación de la macro UNLOCK se realiza a través de una operación de escritura (operación de memoria PrWr) que coloca a cero el valor

de la variable utilizada como semáforo. Como hemos visto en la sección 4.11, las operaciones de memoria PrTAS y PrFAI se realizan exclusivamente sobre datos de un byte, al objeto de evitar situaciones de bloqueo mutuo. Por lo tanto, los semáforos declarados a través de la macro Parmacs LOCKDEC son variables de un byte.

Barreras

La implementación de las barreras se consigue a través de la utilización de la operación de memoria PrFAI, suministrada por el protocolo de memoria de la arquitectura VSR-COMA. Esta operación permite incrementar el contenido de una variable en una única operación atómica, devolviendo a continuación el resultado. La implementación de barreras utilizando dicho mecanismo es sencilla: bastará con incrementar la barrera y comprobar el valor devuelto por la operación. Si el valor devuelto coincide con el valor máximo que se espera de la barrera, el procesador coloca la barrera a 0 y todos los procesadores salen. En caso contrario, el procesador entra en un bucle hasta que el valor de la barrera sea cero.

Este esquema sencillo presenta un serio inconveniente, que surge cuando se utiliza dos veces consecutivas la misma barrera [20]. Cuando el último procesador coloca a 0 la barrera, se produce una escritura que invalida el resto de copias del bloque en las memorias de atracción remotas. Puede darse el caso de que un procesador determinado detecte el final de la barrera, ejecute algunas instrucciones y vuelva a entrar en la barrera antes de que todos los procesadores hayan sido notificados de que durante un cierto tiempo la variable utilizada en la barrera ha estado a cero.

Este problema se soluciona mediante el algoritmo “local sense” [20]. Para utilizar este algoritmo, se define la siguiente estructura para la barrera:

```
struct  _SIM_tipo_barrera {
   char valor;
   char flag;
   char local_sense;
} *Bar;
```

Para modificar el valor de los campos `flag` y `valor` se utilizarán operaciones `PrWr`, ya que la estructura anterior está localizada en el espacio compartido de direcciones. Sin embargo, la variable `local_sense`, pese a formar parte del espacio compartido, no será actualizada a través de operaciones de memoria dirigidas al controlador de coherencia, sino que se utilizará de forma exclusivamente local. La utilización coherente de una variable local situada en el espacio compartido queda en este caso garantizada, ya que se realizará exclusivamente a través de las macros `BARINIT` y `BARRIER`. Esta definición de la estructura de barrera permite agrupar los valores de la barrera y la situación del nodo local respecto de dicha barrera.

El funcionamiento del algoritmo "local sense" es el siguiente. La implementación de la macro BARINIT pone a cero las variables compartidas valor y flag, a través de dos operaciones de memoria PrWr. La variable local_sense también se inicializa a cero.

El algoritmo que utiliza la implementación de la macro BARRIER es el siguiente:

```
Bar->local_sense = ! (Bar->local_sense);
PrFAI (Bar->valor);
if (Bar->valor == valor_maximo) {
   Bar->valor = 0;
   Bar->flag = Bar->local_sense;
}
else
   while (Bar->flag != Bar->local_sense);
```

Cuando se invoque la macro BARRIER, la función que la implementa se encargará de invertir el valor de la variable local_sense: si está a uno se pondrá a cero y viceversa. Como la semántica de la utilización de barreras impide que un proceso pueda alcanzar más de una barrera tras otra, esta variable permite distinguir entre dos pasos consecutivos por la misma barrera [20]. Una vez actualizada local_sense, se realiza una operación PrFAI sobre la variable valor. Si el resultado indica que aún faltan procesos por alcanzar la barrera, el procesador entra en un bucle que se repite hasta que el valor del flag sea el mismo que el de la variable local_sense, indicando que todos los procesos han alcanzado la barrera y que dicha barrera ha sido inicializada para una próxima invocación. El último procesador que alcance la barrera se encargará de poner su valor a cero y de actualizar el flag con el siguiente valor de local_sense, de acuerdo al contenido de su variable local. La implementación de barreras a través de este algoritmo permite una reutilización segura y libre de errores de la macro BARRIER.

5.4.4 Implementación del simulador

Una vez vistas las funciones que implementan los accesos a memoria compartida y las macros Parmacs de sincronización, describiremos brevemente la implementación del simulador EMUCOMA.

Módulos del simulador

La figura 5.3 muestra la estructura de procesos del simulador Emucoma. Como puede verse en la figura, la estructura es muy similar a la propuesta en la fase de diseño (figura 5.2). Uno de los principales objetivos del

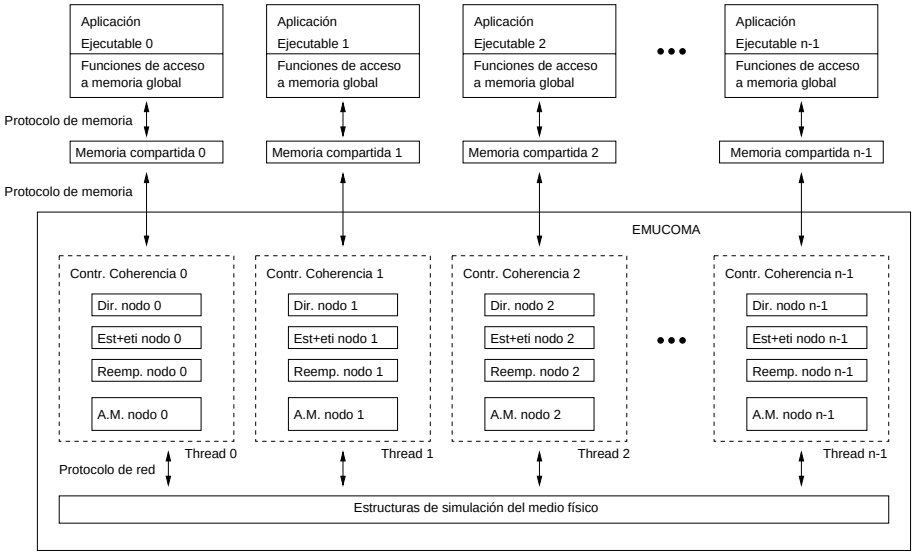


Figura 5.3: Estructura de procesos del simulador EMUCOMA.

sistema propuesto es la reducción del tiempo de simulación. Para ello se desarrolló una arquitectura basada en *threads* [81], en donde cada uno de los controladores de coherencia es un *thread* que mantiene actualizada su propia información. Los *threads* se comunican con los diferentes procesos que forman la aplicación paralela a través de un mecanismo de memoria compartida regulada por semáforos, que describiremos más adelante. Este mecanismo permite que cada uno de los procesos solicite al controlador de coherencia correspondiente la realización de operaciones sobre la memoria compartida, a través del protocolo de memoria de VSR-COMA, descrito en la sección 4.4.1.

A continuación examinaremos brevemente las estructuras de datos que cada controlador de coherencia mantiene actualizadas.

Estructura de las memorias de atracción

La estructura de la memoria de atracción incluye tanto los bloques almacenados como la información de estado y etiqueta. Su estructura es la siguiente:

```
typedef struct {
   Tmarco marco[G_NUM_MARCOS][G_TAM_MARCO];
   Tetiqueta etiqueta[G_NUM_MARCOS];
   Testado estado[G_NUM_MARCOS];
} Tconjunto;
```

```
typedef struct {
   Tconjunto cjto[G_NUM_CJTOS_AM];
} Tmem_atrac;
```

La estructura `Tconjunto` permite almacenar los bloques que componen el conjunto, junto con sus estados y etiquetas asociados. La memoria de atracción de cada controlador está formada por un vector de dichos conjuntos.

Esta definición de la memoria de atracción permite que cada uno de los controladores de coherencia reserven el espacio destinado a ella a través de una única llamada a la función `malloc()`, lo que reduce la complejidad de la reserva de memoria.

Estructura de los directorios

La información de directorio mantenida por cada controlador de coherencia se almacena en un vector que indica el propietario de cada uno de los bloques presentes en el sistema. Al igual que lo que ocurre en el caso de las memorias de atracción, cada *thread* reserva espacio para su vector de directorio.

Estructura de la información de reemplazo

Las estructuras que permiten almacenar la información de reemplazo son las siguientes:

```
typedef struct {
   Tetiqueta etiqueta[G_NUM_MARCOS];
   Testado estado[G_NUM_MARCOS];
} Tcjtoreemp;

typedef struct {
   Tcjtoreemp cjto[G_NUM_CJTOS_AM];
} Treemp;
```

De esta manera, se define una estructura `Tcjtoreemp` que contiene la información de reemplazo para un conjunto determinado. La estructura `Treemp` almacena la información de reemplazo para todos los conjuntos de un nodo. Para utilizar ambas estructuras, cada *thread* define un vector de n punteros, cada uno de ellos destinado a almacenar la información de reemplazo de un nodo remoto. A través de $n-1$ llamadas a `malloc()` se reserva memoria para la información de reemplazo de los $n-1$ nodos remotos. El elemento del vector cuyo índice coincide con el identificador del nodo actual no se utiliza.

Estructura de simulación del medio físico

El envío y recepción de eventos del protocolo de coherencia se realiza a través de una estructura de simulación del medio físico, consistente en la utilización de un vector de eventos enviados. El simulador de arquitectura mantiene un único contador que indica cual es el número del último evento escrito en la red, y n contadores que indican el número del último evento leído por cada uno de los controladores de coherencia. Cada controlador de coherencia puede solicitar la escritura de un nuevo evento y la lectura de un evento pendiente a través de un par de funciones que actúan como interfaz. Se garantiza la exclusión mutua en el acceso a través de un conjunto de semáforos para *threads*, los cuales conllevan un coste de ejecución menor que los semáforos utilizados en la sincronización de procesos. La utilización de un vector de eventos permite centralizar la escritura de nuevos eventos y a la vez mantener una única cola de eventos producidos, lo que disminuye en gran medida la memoria ocupada por los *threads* que simulan el funcionamiento de los controladores de coherencia, en especial al aumentar el número de nodos.

La estructura del vector que simula el funcionamiento de una red de bus común es la siguiente:

```
typedef struct s_evento{
   Tevento tipoevento;
   Tnodo origen;
   Tnodo destino;
   Tmarco num_marco;
   Tdescriptor descriptor;
   Tnumevento num_evento;
   Tbloque bloque[G_TAM_MARCO];
} Teventoenred ;

Teventoenred G_cola_eventos[EVENTOS_MAX];
```

El correcto funcionamiento de este mecanismo de simulación del medio físico depende de que todos los eventos lleguen a todos los nodos, sin que se produzca ninguna clase de pérdidas debido a que un controlador no haya podido procesar todos los eventos de entrada. El tamaño del vector descrito arriba (determinado por la constante EVENTOS_MAX) debe elegirse de modo que sea suficientemente grande para garantizar la existencia simultánea de todos los eventos no procesados, sin que su tamaño sea excesivo. En las simulaciones se ha determinado que un valor de 10.000 es suficiente para este cometido. Por otra parte, se han desarrollado mecanismos de protección para el caso de que el número de eventos no procesados supere el tamaño establecido para la estructura de almacenamiento.

Comunicaciones y sincronización

Como puede verse en la figura 5.3, existen básicamente dos rutas diferentes para el intercambio de datos en la simulación: la utilizada para la comunicación entre los controladores de coherencia, realizada a través de eventos del protocolo VSR-COMA, y la utilizada por cada proceso para comunicarse con su controlador correspondiente, al objeto de acceder a la memoria compartida a través del protocolo de memoria.

Como se ha descrito en el apartado anterior, los eventos de protocolo intercambiados por los controladores de coherencia se transmiten a través de una estructura de datos común a todos ellos: la estructura de simulación del medio físico. Respecto al mecanismo de comunicaciones entre cada uno de los procesadores y su controlador de coherencia, se ha implementado un mecanismo basado en el uso de memoria compartida [80]. La estructura de datos intercambiada entre ambos a través de la memoria compartida es la siguiente:

```
typedef struct s_operacion{
   Tbool nueva;
   Ttipooper opcode;
   Tdireccion dir;
   Ttam tam;
   char dato[G_TAM_BYTES_DATO_MAXIMO];
} Toperacion;
```

El campo `nueva` se utiliza para que el controlador de coherencia determine si la operación presente en la zona de memoria compartida es nueva o ya ha sido resuelta. Cuando cada proceso solicita una nueva operación, coloca este campo a “true”: el controlador de coherencia cambia su valor a “false” al leer dicha solicitud. El campo `opcode` identifica la operación solicitada por el procesador al controlador de coherencia. Además de las operaciones de memoria soportadas por el protocolo VSR-COMA y descritas en la sección 4.4.1, se han añadido otras tres operaciones para controlar el funcionamiento del controlador de coherencia:

PrStart : esta operación se utiliza para indicar al controlador de coherencia que inicie la contabilidad de los accesos. Se envía al controlador al inicio de la zona paralela del código, al objeto de no contar los accesos a la memoria compartida debidos a la inicialización de los datos, que constituye una etapa previa al funcionamiento en paralelo de los procesos.

PrStop : esta operación se utiliza para indicar al controlador de coherencia que detenga la contabilidad de los accesos a la memoria compartida. Por lo tanto, todos los accesos subsiguientes no serán tenidos en cuenta a dicho efecto.

PrEnd : esta operación se utiliza para indicar al controlador de coherencia que el proceso correspondiente ha finalizado la ejecución de su parte de la aplicación. Ningún controlador de coherencia podrá terminar hasta que *todos* los procesos hayan acabado, ya que cualquiera de los restantes procesos pueden solicitar un dato presente en la memoria de atracción de cualquier nodo remoto.

Los restantes campos de la estructura vista sirven para indicar la dirección del dato que desea modificarse, su tamaño y el buffer que se utilizará para la lectura o escritura correspondiente.

Cabe destacar aquí que la aplicación comunica al controlador el acceso a realizar utilizando exclusivamente esta estructura. El controlador deberá determinar, en función de la arquitectura, si la operación solicitada requiere la actualización de uno o dos bloques de la memoria compartida. El tamaño máximo del dato al que podrá accederse en una única operación está limitado en EMUCOMA por la constante `G_TAM_BYTES_DATO_MAXIMO`, que debe ser siempre menor que el tamaño del bloque utilizado en el mantenimiento de la coherencia de las memorias de atracción. Este hecho impide que una única solicitud genere más de dos faltas de bloque. Cabe destacar que el tamaño máximo que puede transferirse mediante una única operación de memoria depende de la arquitectura sobre la que se ejecuta la aplicación, por lo que dicho tamaño debe conocerse con anterioridad para asegurar el correcto funcionamiento del simulador EMUCOMA, a través de la asignación de valores a la constante ya mencionada.

Para la lectura y escritura de la zona de memoria compartida se utilizan dos semáforos. Uno de ellos se utiliza para solicitar el acceso en exclusión mutua a los datos compartidos, tanto para lectura como para escritura. Este semáforo impide que la estructura sea modificada por la aplicación mientras el controlador de coherencia la está leyendo para comprobar si existe una nueva solicitud. El segundo semáforo se utiliza para suspender la ejecución del proceso que forma parte de la aplicación paralela hasta que el acceso a la memoria compartida se haya resuelto. Una vez que el proceso solicita una nueva operación, ejecuta una operación de solicitud de recurso sobre dicho semáforo (a través de una primitiva del tipo `SEM_P`). Esto hace que el proceso se suspenda. Cuando el controlador de coherencia haya resuelto la solicitud del proceso, realizará una operación `SEM_V` sobre dicho semáforo, haciendo que el proceso se reanude, recibiendo la respuesta a la operación de memoria suministrada por el controlador de coherencia. El objetivo de este sistema es evitar la espera activa por parte del proceso, permitiendo así que el *thread* que representa al controlador de coherencia indique cuándo puede continuar la ejecución del mismo, lo que reduce el consumo de CPU y mejora la velocidad de ejecución de la simulación.

Las memorias de atracción y el espacio de direcciones compartido

Una vez descrita la arquitectura del simulador, nos queda por examinar el papel que juega el tamaño de cada una de las memorias de atracción en la simulación. La presión de memoria (el cociente entre el tamaño del espacio de direcciones compartido y la suma de los tamaños de las memorias de atracción) se regula modificando el tamaño asignado a cada memoria de atracción. En la práctica esto se consigue aumentando o disminuyendo el número de conjuntos de cada AM en función del tamaño del espacio de direcciones compartido y del número de procesadores.

El dimensionamiento de las memorias de atracción plantea algunas cuestiones. En primer lugar, los experimentos sobre mejoras en el rendimiento al aumentar el número de procesadores suponen una presión de memoria constante. Esto obliga a recalcular el tamaño que debe tener cada AM en función del número de procesadores sobre el que va a realizarse la ejecución de la aplicación paralela. Pero para que la comparativa tenga sentido, es fundamental que el tamaño del espacio de direcciones compartido sea constante e independiente del número de procesadores. Aunque habitualmente esto se cumple [83], si dicho tamaño variara al cambiar el número de procesadores, esto obligaría a modificar también la capacidad total de las memorias de atracción para mantener la presión de memoria constante. En consecuencia, cambiará el número total de bloques libres para replicación, haciendo variar también la distribución de accesos entre los diferentes conjuntos. Por lo tanto, un cambio en el tamaño del espacio de direcciones compartido puede modificar completamente el patrón de accesos a la memoria compartida.

Si la aplicación utiliza un espacio de direcciones compartido de tamaño dependiente del número de procesadores, lo que como hemos visto no es en absoluto deseable con vistas a evaluar su rendimiento, una forma de minimizar los efectos producidos por el cambio es utilizar un espacio compartido de memoria con un tamaño igual al mayor de los espacios de direcciones compartidos que la aplicación vaya a utilizar. De esta manera, cuando se ejecute la aplicación sobre un espacio de direcciones compartido menor, debido a un cambio en el número de procesadores, lo único que ocurrirá será que habrá algunos bloques del espacio compartido a los que nunca se accederá, pero que están presentes, lo que mantiene constante el número de marcos de bloque disponibles para replicación y minimiza los efectos provocados por cambios en el espacio de direcciones compartido .

Otra cuestión importante es la que atañe a la distribución inicial del espacio de direcciones compartido entre las diferentes memorias de atracción. Una de las principales ventajas de las arquitecturas COMA es la posibilidad que tienen los bloques de migrar hacia los nodos que los estén utilizando. Por lo tanto, la distribución inicial no afecta tanto al rendimiento como en otras arquitecturas de memoria compartida distribuida. Dado

que los sistemas COMA no conocen a priori el patrón de accesos de la aplicación a la memoria compartida, no se dispone de información alguna que permita distribuir el almacenamiento del espacio de direcciones compartido entre las diferentes memorias de atracción. En consecuencia, la solución adoptada en el simulador EMUCOMA para asignar los bloques a las diferentes AM consiste en determinar el conjunto al que pertenece cada bloque y almacenarlo en el primer nodo que tenga espacio libre en dicho conjunto, contando desde el nodo 0. Una vez concluida la inicialización de estructuras en el simulador de arquitectura, todos los bloques están distribuidos entre las memorias de atracción, habiendo sido actualizada además la información de directorio y la de reemplazo. Cuando el nodo encargado de las inicializaciones comience a acceder a los bloques de la memoria compartida, estos accesos provocarán faltas de capacidad en los conjuntos de dicho nodo, obligando a efectuar reemplazos hacia los nodos con marcos de bloque libres siguiendo el mecanismo de selección de nodo destino de VSR-COMA. En consecuencia, la carga se distribuirá entre las memorias de atracción.

Funcionamiento general de los controladores de coherencia

El funcionamiento de los controladores de coherencia sigue las pautas descritas en la sección 4.6. Sólo se procesan operaciones de memoria cuando no haya eventos pendientes de procesamiento. Por otra parte, hasta que una operación de memoria no es resuelta no se inicia el tratamiento de la siguiente.

El algoritmo utilizado por los controladores de coherencia se complica ligeramente al considerar la posibilidad de que la solicitud de memoria obligue a acceder a dos bloques que pueden estar o no presentes en la memoria de atracción local. Si al menos uno de los bloques está presente en la AM local, la parte de la operación que le corresponde se realiza inmediatamente, para evitar que mientras que se solicita el bloque ausente dicho bloque sea requerido por un nodo remoto, lo que obligaría a volverlo a pedir, generándose así otra falta cache. El orden en el que se realice la actualización de un dato situado en los límites de dos bloques es indiferente: se supone que si dicho dato puede ser accedido simultáneamente por dos nodos, el programador lo habrá situado dentro de una zona de exclusión mutua, garantizándose así la consistencia en la actualización.

Simulación de mecanismos de reemplazo

El simulador EMUCOMA utiliza el protocolo VSR-COMA para el mantenimiento de la coherencia entre las memorias de atracción que componen el sistema. Sin embargo, gracias a la separación existente en VSR-COMA entre el protocolo de coherencia y la selección del nodo destino de una ope-

ración de reemplazo, cabe la posibilidad de utilizar el simulador EMUCOMA con diferentes mecanismos de selección, lo que permite comparar el comportamiento de las diferentes estrategias de reemplazo sin modificar el protocolo. Este punto es de especial importancia en el estudio, ya que utilizar diferentes protocolos lleva a resultados difícilmente comparables, al ser distintas las condiciones de ejecución.

Para comparar el funcionamiento de los distintos algoritmos de reemplazo descritos en el capítulo 3, se han desarrollado cuatro módulos de reemplazo distintos para su utilización en EMUCOMA:

Reemplazo aleatorio : este mecanismo selecciona aleatoriamente el nodo destino de una operación de reemplazo. Si el nodo rechaza la solicitud de reemplazo, se elige aleatoriamente un nodo entre los restantes. Este mecanismo es el utilizado por COMA-F [45], descrito en la sección 3.3.1.

Reemplazo por consulta : en este algoritmo, el reemplazo se realiza una vez consultado el estado del conjunto correspondiente en todas las memorias de atracción. Este es el mecanismo utilizado en DICE [15], descrito en la sección 3.3.2.

Estrategia de reemplazo de VSR-COMA : se utiliza la estrategia descrita para VSR-COMA, y basada en el almacenamiento de la información de estado y etiqueta para todos los conjuntos remotos.

Sistema sin reemplazo : para poder evaluar correctamente la importancia de las mejoras obtenidas en el rendimiento, se ha desarrollado una versión sin reemplazo, en la que la capacidad de cada una de las memorias de atracción es igual al espacio compartido de direcciones. Este enfoque es el utilizado en COMA-BC [84].

La utilización de los cuatro módulos de reemplazo distintos nos ha permitido comparar el rendimiento de los algoritmos de reemplazo en la ejecución de aplicaciones paralelas. Los resultados obtenidos se presentan en el capítulo 6.

5.5 Índices de rendimiento utilizados

Para poder comparar el funcionamiento de diferentes arquitecturas en la ejecución de aplicaciones paralelas se necesita información acerca de cómo se ha desarrollado dicha ejecución: número de instrucciones ejecutadas, número de accesos a memoria y de eventos de protocolo generados por la arquitectura. El sistema de simulación descrito permite obtener dos clases de índices: los referidos a la aplicación y los de la arquitectura COMA subyacente.

Las funciones encargadas de la contabilidad de la aplicación se encuentran entre las funciones de biblioteca que se enlazan con el fichero objeto de dicha aplicación. Entre los índices más importantes que proporcionan se encuentran los siguientes:

Instrucciones (i): indica el número total de instrucciones ejecutadas por la aplicación. Estos índices se obtienen instrumentalizando el código en ensamblador de la aplicación, de forma que tras la ejecución de cada instrucción se invoca a una función que incrementa un contador. Como se ha visto en la sección 5.3.4, la instrumentalización de la aplicación se realiza antes del enlace. Por lo tanto, el recuento de instrucciones no incluye las instrucciones de las funciones de comunicaciones con el controlador de coherencia ni las de sincronización, sino sólo las referidas a la aplicación.

Accesos a memoria (m): se almacena el número de accesos a memoria para lectura (m_r) y escritura (m_w), tanto a la memoria compartida como a la memoria local de cada proceso.

Accesos a memoria compartida (s): se almacena el número de accesos a la memoria compartida para lectura (s_r) y escritura (s_w). A estos efectos, los accesos para sincronización (a través de operaciones PrTAS y PrFAI) se consideran accesos en escritura.

Por otra parte, los índices de rendimiento referidos a la arquitectura simulada son responsabilidad del simulador de arquitectura. Además de posibilitar la ejecución de aplicaciones paralelas que utilicen memoria compartida, el simulador EMUCOMA se encarga de mantener el recuento de solicitudes de lectura y escritura por parte de los procesos y de los eventos producidos por cada controlador de coherencia. Los eventos generados se asignan a los controladores de coherencia de la siguiente manera. Las solicitudes que un controlador genere se asignan siempre a ese controlador, mientras que los eventos de respuesta se asignan al controlador que ha realizado la solicitud. Esto permite determinar el número de eventos que ha tenido que transmitir y recibir cada controlador para resolver las solicitudes de acceso al espacio compartido que haya recibido. Todos los eventos que un controlador genera en respuesta a solicitudes remotas se contabilizan en el nodo que ha iniciado la transacción.

Los índices suministrados por el simulador de arquitectura al final de la ejecución de la aplicación paralela son los siguientes:

Accesos a memoria compartida (s): número de solicitudes de lectura (s_r) y escritura (s_w) recibidas. Estos índices deben necesariamente coincidir con los índices correspondientes de la aplicación.

Accesos a bloques (b): número de bloques de la memoria de atracción accedidos, tanto para lectura (b_r) como para escritura (b_w). Su número deberá ser mayor o igual que el número de accesos, ya que una referencia a memoria compartida puede suponer el acceso a más de un bloque.

Número de accesos locales (ba): número de accesos a bloques que se han resuelto de forma local, tanto para lectura (ba_r) como para escritura (ba_w).

Número de accesos remotos (bf): número de accesos a bloques que se han resuelto de forma remota, tanto para lectura (bf_r) como para escritura (bf_w).

Eventos debidos a lecturas y a escrituras remotas (e): número de eventos producidos por las lecturas (e_r) y las escrituras (e_w) remotas. Se incluyen los eventos debidos a operaciones de reemplazo necesarias para completar los accesos remotos.

Número de bytes por evento (l_m): promedio de bytes por cada uno de los eventos generados. Representa la longitud media de los eventos intercambiados en la red.

Los índices así obtenidos se utilizarán para comparar el rendimiento de diferentes arquitecturas en la resolución de un mismo problema, a través de un modelo analítico. En la siguiente sección describiremos un modelo analítico del sistema que nos permitirá utilizar los índices de rendimiento para determinar la mejora en la velocidad de ejecución (*speedup*) del sistema. Este modelo es el que utilizaremos en el capítulo 6 para la evaluación comparativa de rendimientos.

5.6 Análisis de rendimientos empleando el modelo LogP

Se ha desarrollado un sencillo modelo analítico que permite obtener resultados sobre el rendimiento en la ejecución de aplicaciones paralelas sobre un sistema VSR-COMA, partiendo de los resultados devueltos por el simulador EMUCOMA. Como hemos visto en la sección 5.4.1, el simulador EMUCOMA simula la ejecución de una aplicación paralela en un conjunto de n nodos unidos por una red de tipo bus. Para construir nuestro modelo analítico hemos considerado un sistema VSR-COMA en el que los nodos son estaciones de trabajo físicamente idénticas, unidas por una red de bus común. En dicho modelo analítico existen unos datos de entrada y unos datos de salida. Los datos de entrada se pueden dividir en dos subconjuntos:

1. Resultados obtenidos en la simulación multiprocesador a través de la colección de índices de rendimiento descritos en la sección 5.5.

2. Parámetros que describen las características físicas del sistema VSR-COMA concreto. Dentro del conjunto de estos últimos podemos distinguir a su vez dos grupos:

   (a) Parámetros que describen las características físicas del procesador, que son la duración del ciclo de reloj del procesador (τ) y el número de ciclos necesario por término medio para completar cada instrucción.

   (b) Parámetros que describen el comportamiento de la red de interconexión de las estaciones de trabajo, los cuales han sido escogidos representando el comportamiento del sistema de interconexión por medio del modelo LogP [18, 19].

El modelo analítico permite obtener como salida el tiempo que tardaría en ejecutarse la aplicación paralela sobre un conjunto de estaciones de trabajo con un número de procesadores igual al considerado en la simulación y con las características físicas establecidas en los parámetros antes mencionados. A partir de este resultado es inmediato obtener el incremento en la velocidad de ejecución (*speedup*) para distinto número de procesadores.

Para calcular el tiempo total de ejecución de la aplicación paralela consideramos como tal el tiempo transcurrido desde el inicio de la ejecución del código paralelo, que se considera simultáneo para todos los procesadores, hasta el instante en el que termina dicha ejecución. Como las distintas estaciones de trabajo pueden ejecutar partes de la carga no exactamente del mismo tamaño, se considera como tiempo de ejecución $T(n)$ en un conjunto de n estaciones de trabajo al mayor de los tiempos calculados para las diversas estaciones de trabajo (nótese que los índices devueltos por el simulador, descritos en la sección 5.5, se refieren a cada una de los procesadores involucrados).

El modelo analítico obtiene el tiempo total de ejecución en un conjunto de n estaciones de trabajo como la suma de dos términos:

$$T(n) = t_c + t_r$$

En dicha expresión t_c representa el tiempo consumido para ejecutar toda la parte de carga de trabajo que no hace referencia a la comunicación a través de la red. Frente a ello, t_r representa el tiempo involucrado en las comunicaciones a través de la red necesarias para ejecutar esa carga de trabajo.

Para calcular t_c se considera que el tiempo consumido en la ejecución de cada instrucción se compone de tres partes:

1. Un tiempo fijo para toda instrucción, al que denominamos t_i.
2. Un tiempo extra fijo t_m si se trata de una instrucción de lectura o de escritura en la memoria.
3. Otro tiempo extra fijo t_s para el caso de que se trate de una instrucción de lectura o escritura en la zona de memoria compartida.

Con ello tenemos que:

$$t_c = it_i + mt_m + st_s = (ic_i + mc_m + sc_s)\tau$$

Esos tres términos c_i, c_m y c_s representan los tiempos t_i, t_m y t_s expresados en número de ciclos de reloj del procesador. Constituyen los tres parámetros de entrada del modelo que describen el número de ciclos necesarios para completar cada instrucción, y que deben ser ajustados (al igual que τ) en función de la arquitectura física de las estaciones de trabajo utilizadas.

Para calcular t_r, se considera que ese tiempo va a incluir todos los tiempos necesarios para completar, utilizando la red, todas las instrucciones de lectura o escritura que se hayan realizado sobre el espacio compartido y que hayan dado lugar a una falta, esto es, una referencia a memoria que no haya podido ser completada sin acceder a la red. Obtenemos t_r simplemente como

$$t_r = ew$$

donde w representa el tiempo promedio involucrado en la transmisión por la red de un evento o mensaje.

Para calcular w utilizamos una representación de la red basada en un modelo LogP. En concreto se ha supuesto que la frecuencia de mensajes es lo suficientemente pequeña como para poder despreciar el término de tiempo entre mensajes sucesivos enviados a través de la red ("gap") [47], con lo que w se puede expresar como:

$$w = o_s(l_m) + o_r(l_m) + L(l_m) = a + bl_m$$

siendo $o_s(l)$, $o_r(l)$ y $L(l)$ respectivamente el coste fijo de envío, el coste fijo de recepción y la latencia de la red de un mensaje de longitud l bytes. Los términos a y b son los parámetros de entrada del modelo que describen el comportamiento del mecanismo de interconexión. El término l_m representa la longitud media de los mensajes intercambiados, como ya hemos comentado.

El modelo analítico descrito será utilizado para comparar el rendimiento de un sistema multicomputador con gestión de memoria de tipo COMA en la ejecución de un conjunto de programas paralelos. En el capítulo siguiente se presentan los resultados obtenidos.

5.7 Conclusiones

Existen diferentes técnicas para estudiar el funcionamiento de un protocolo de coherencia para un sistema de memoria compartida distribuida. La técnica utilizada en el presente trabajo es la de la simulación basada en ejecución. Para ello se ha desarrollado un simulador denominado EMUCOMA, que permite la ejecución de aplicaciones paralelas utilizando el protocolo VSR-COMA para el mantenimiento de la coherencia del espacio de memoria compartido. La utilización del simulador EMUCOMA requiere la aplicación de un modelo de programación paralelo basado en el uso de macros Parmacs y de algunas convenciones a la hora de acceder al espacio de memoria compartido, ambas descritas en el presente capítulo. Para poder aplicar los índices de rendimiento devueltos por el simulador EMUCOMA se ha desarrollado además un modelo analítico que establece el coste asociado a la ejecución de las aplicaciones en un sistema multicomputador formado por un conjunto de estaciones de trabajo y unidos por una red de bus común. La utilización conjunta del simulador EMUCOMA y del modelo analítico descrito en la sección 5.6 permite evaluar el rendimiento de un sistema multicomputador basado en el protocolo VSR-COMA en la ejecución de programas paralelos.

En el capítulo siguiente se presentan los resultados obtenidos de la utilización del simulador en la ejecución de diferentes programas paralelos. Como veremos, el sistema de simulación no sólo permite obtener los índices de rendimiento para el protocolo VSR-COMA con su estrategia de reemplazo a diferentes presiones de memoria, sino también comparar los resultados obtenidos con los índices resultantes de utilizar los otros mecanismos de reemplazo descritos.

Capítulo 6

Evaluación comparativa de rendimientos

6.1 Introducción

En los dos capítulos anteriores se han descrito el protocolo VSR-COMA y el simulador basado en ejecución EMUCOMA desarrollado para estudiar el comportamiento de dicho protocolo. En este capítulo se presentarán los resultados de rendimiento obtenidos. La sección 6.2 describe las características de los programas utilizados como cargas de trabajo, todas ellas pertenecientes al conjunto de programas SPLASH-2. La sección 6.3 presenta los resultados obtenidos para la ejecución de dichos programas utilizando el protocolo VSR-COMA y su mecanismo de reemplazo asociado. Los resultados se presentan en términos de aceleración (*speedup*) y en tráfico total generado en la red de interconexión.

En la sección 6.4 se compara el funcionamiento de la estrategia de reemplazo de VSR-COMA con las otras estrategias de selección de nodo destino descritas en el capítulo 3: la selección aleatoria y la selección basada en consulta. Estos resultados se comparan a su vez con un sistema VSR-COMA sin reemplazo, para poder así mejor valorar la importancia relativa del mecanismo de reemplazo en la mejora del rendimiento.

La mejora en el rendimiento producida a través del mecanismo de reemplazo que utiliza VSR-COMA se obtiene a costa de mantener una mayor cantidad de información de control en cada controlador de coherencia. En la sección 6.5 se cuantifica ese incremento para los programas utilizados.

6.2 Características de la carga de trabajo

Para la evaluación de rendimientos del protocolo VSR-COMA se escogieron programas pertenecientes al conjunto SPLASH-2 [11]. Entre las principales ventajas que reporta su uso podemos citar las siguientes:

- Su aceptación como cargas de trabajo útiles para el estudio de sistemas multiprocesadores de memoria compartida, al tratarse de problemas representativos de aplicaciones de cálculo científico intensivo.

- La existencia de bibliografía que describe en detalle las propiedades fundamentales de cada programa, lo que permite comprender mejor la utilización que cada uno de ellos hace de la memoria compartida.

- La posibilidad de dimensionar los problemas a resolver en cada programa, lo que permite adaptar el problema a las características del sistema de simulación. Esto permite obtener un balance adecuado entre la representatividad del problema y las limitaciones en las capacidades de cálculo del sistema de simulación.

- Finalmente, cabe citar la política de libre distribución del código de los programas que componen el conjunto Splash-2. Esto permite disponer de un conjunto de programas ampliamente utilizado para la comparación de rendimientos sin coste alguno.

Para la evaluación de rendimientos del sistema VSR-COMA se han escogido un total de seis programas: tres *kernels* (FFT, LU y Radix) y tres aplicaciones (Ocean, Barnes y Radiosity). Cada uno de estos programas ha sido convenientemente modificado para su uso con EMUCOMA de acuerdo al modelo de programación descrito en la sección 5.3. A continuación describiremos brevemente las características de cada uno de estos programas.

6.2.1 FFT

El *kernel* FFT consiste en el cálculo de la transformada rápida de Fourier sobre un conjunto de n datos complejos, optimizado para minimizar la comunicación entre procesos. El conjunto de datos consiste en los n datos complejos a transformar, y otros n puntos complejos, referidos como las *raíces de unidad* [11]. Ambos conjuntos de datos están organizados como matrices de tamaño $\sqrt{n}$ x $\sqrt{n}$ divididas de forma que se asigna a cada procesador un conjunto de filas consecutivas. Las comunicaciones se producen en la fase de transposición, lo que requiere comunicaciones entre todos los nodos [102]. El *kernel* FFT calcula la transformada del vector y a continuación su transformada inversa, computándose luego la diferencia entre el vector obtenido y el original, con el objeto de detectar eventuales errores.

Los experimentos que hemos realizados con el *kernel* FFT han utilizado un valor para n de 65536. Cabe destacar que el tamaño del espacio de direcciones compartido en FFT cambia ligeramente con el número de procesadores, con un tamaño de 3100 Kb para un único procesador e incrementándose a razón de algo más de 12 Kb por procesador en los restantes experimentos. Los efectos que esta modificación del tamaño del espacio

compartido de direcciones pudieran tener en la evaluación de rendimientos han sido minimizados por el procedimiento descrito en la sección 5.4.4. Se ha elegido como tamaño para el espacio de direcciones compartido de FFT el correspondiente a 16 procesadores (3300 Kb), y en relación a este tamaño se han calculado los tamaños correspondientes a las memorias de atracción, según el número de nodos y la presión de memoria deseada en cada caso.

6.2.2 LU

El *kernel* LU realiza la factorización de una matriz densa, a través del producto de una matriz triangular superior por una matriz triangular inferior. La matriz densa A se divide en un vector de tamaño N x N de B x B bloques, al objeto de aprovechar la localidad *temporal* [11] de los elementos de la submatriz. Los elementos dentro de un bloque se almacenan de forma contigua, al objeto de aprovechar la localidad *espacial* [102]. Las matrices obtenidas se multiplican para comparar el resultado con la matriz original, asegurándose la corrección del funcionamiento del *kernel*.

El tamaño del problema elegido es el de una matriz de 256 x 256 elementos, con un tamaño de bloque $B = 16$. El tamaño del espacio de direcciones compartido resultante es de 640 Kb.

6.2.3 Radix

El *kernel* Radix es un programa de ordenación iterativo [6, 106]. En cada iteración se procesan r claves. Cada procesador procesa sus claves, generando un histograma con las mismas. Este histograma local permite asignar las claves a los nodos remotos que las procesarán en la siguiente iteración. Tras acumularse todos los histogramas locales en un único histograma global, los procesadores lo utilizan para permutar sus claves e iniciar una nueva iteración, lo que requiere comunicaciones entre todos los nodos [101, 39].

Los valores que hemos asignado al problema fueron la ordenación de 2^{20} claves, con un valor $r = 1024$. La resolución de este problema trabaja con un espacio de direcciones compartido de 9 Mb.

6.2.4 Ocean

Esta aplicación estudia los movimientos de masas oceánicas a gran escala, basadas en corrientes de frontera y de remolino. Se trata de una versión mejorada de la aplicación Ocean de Splash [77]. El tamaño del problema base según los autores es una malla de 258 x 258 kilómetros, y es el tamaño que hemos elegido para nuestra simulación. El tamaño del espacio de direcciones compartido resultante es de 16 Mb.

6.2.5 Barnes

Esta aplicación se encarga de simular la interacción de un sistema de cuerpos en tres dimensiones a lo largo de un número de pasos que representan unidades de tiempo. Para ello utiliza el método jerárquico de Barnes-Hut para N cuerpos [92]. Al igual que sucede con la aplicación Ocean, Barnes supone una evolución de su homónima del conjunto Splash inicial [77]. El patrón de comunicaciones depende de la distribución inicial de las partículas [11].

El tamaño del problema escogido es de 4096 partículas, lo que conlleva un espacio de direcciones compartido de 5 Mb.

6.2.6 Radiosity

Esta aplicación calcula una distribución equilibrada de la luz en una escena utilizando el algoritmo de irradiación difusa jerárquico e iterativo [76]. La escena se modela inicialmente como un número de polígonos de entrada. Se calcula las interacciones lumínicas entre los polígonos, dividiéndolos en polígonos más pequeños cuando sea necesario mejorar la exactitud del resultado. En cada iteración, el algoritmo opera sobre un conjunto de piezas, subdividiendo dichas piezas recursivamente. Al final de cada iteración, se calcula la irradiación producida por cada pieza al objeto de comprobar si la irradiación general converge hacia un cierto valor.

El paralelismo en el sistema se gestiona a través de colas de tareas, a razón de una por procesador, utilizando una técnica de "robo de tareas" para mejorar la distribución de las mismas. Cada procesador consume tareas de su propia cola. Cuando dicha cola queda vacía, el procesador roba tareas de otras colas. Para minimizar los efectos de pérdida de localidad espacial que se produce al robar tareas de otros procesadores se utiliza el siguiente método: cada procesador inserta las nuevas tareas generadas recursivamente al principio de su cola de tareas, mientras que las tareas robadas se toman del final de las colas de otros procesadores. De esta manera se tiende a robar tareas de piezas grandes, lo que mejora la localidad espacial del problema a resolver en cada procesador.

La aplicación Radiosity propone tres escenas como problemas base: una sencilla escena llamada TEST, que permite comprobar el correcto funcionamiento del algoritmo, y dos escenas denominadas ROOM y LARGE-ROOM, de las cuales hemos utilizado la primera. La escena ROOM supone la utilización de un espacio de direcciones compartido de 32 Mb. El resto de parámetros que hemos utilizado en la ejecución de Radiosity son los mismos que se utilizan en la bibliografía [76].

Programa	Tamaño del problema	Memoria compartida	Instr. ($x10^6$)
LU	Matriz de 256 x 256	640 Kb	231
FFT	65.536 puntos	3300 Kb	205
Radix	1.048.576 puntos	9 Mb	663
Ocean	Malla de 258 x 258 km	16 Mb	1.427
Barnes-Hut	4096 partículas	5 Mb	1.802
Radiosity	Modelo "ROOM"	32 Mb	4.090

Tabla 6.1: Características de los programas Splash-2 utilizados en las simulaciones.

6.2.7 Resumen de características de los programas

La tabla 6.1 muestra un resumen de las características de los programas Splash-2 descritos más arriba. Además de la información referente al tamaño del espacio de direcciones compartido, ya descrita más arriba, se indica el número aproximado de instrucciones de cada programa cuando se ejecuta en un único nodo.

6.3 Rendimiento de VSR-COMA

A continuación se estudiará el rendimiento de VSR-COMA al ejecutar los programas descritos en la sección anterior bajo diferentes presiones de memoria. Las ejecuciones de los programas en nuestro simulador EMUCOMA nos han permitido obtener los incrementos en el rendimiento bajo tres valores de presión de memoria diferentes (25%, 50% y 80%), utilizando memorias de atracción asociativas de 8 vías, bloques de 256 bytes y un número de procesadores entre 1 y 16.

El límite superior en las curvas de rendimiento viene expresado por el comportamiento del protocolo en el caso de que las memorias de atracción tuvieran un tamaño suficiente como para almacenar todos los bloques accedidos, sin necesidad de generar ninguna operación de desalojo. Por este motivo, compararemos los resultados obtenidos para las tres presiones de memoria señaladas con los que se obtienen con un sistema VSR-COMA que utilice para cada AM un tamaño igual al espacio de direcciones compartido de la aplicación. Este comportamiento es similar al propuesto en COMA-BC [84], aunque los rendimientos aquí obtenidos son algo mejores ya que se utilizan mecanismos de sincronización basados en operaciones atómicas sobre datos del espacio compartido, en lugar de los algoritmos de exclusión mutua usados en COMA-BC, lo que genera un menor número de faltas remotas en las operaciones de sincronización.

Los resultados obtenidos se basan en los índices de rendimiento devueltos tras cada ejecución por el simulador EMUCOMA, detallados en la sec-

ción 5.5. Estos índices se han aplicado al modelo analítico de rendimiento descrito en la sección 5.6, utilizando algunos parámetros adicionales para describir las características de la arquitectura de cada estación de trabajo y de la red de interconexión. En concreto, se ha supuesto la utilización como nodos de proceso de estaciones de trabajo de tipo RISC a 167 MHz, lo que supone un valor para τ de 7 ns. De acuerdo con los datos obtenidos de arquitecturas estándar se han considerado los siguientes valores para el resto de parámetros de descripción del sistema: un tiempo t_i fijo para cada instrucción de 1 ciclo (7 ns), un tiempo extra fijo t_m de acceso a memoria de 2 ciclos (14 ns), y un tiempo extra fijo t_s de acceso a la memoria compartida de 10 ciclos (140 ns). En cuanto al medio de interconexión se ha supuesto una red tipo Myrinet utilizando "fast messages" [75], a la que según la bibliografía [7, 47, 28, 67] se han asignado los parámetros de entrada para el modelo analítico $a = 10\mu s$ y $b = 0.026\mu s$.

Los resultados aquí presentados son básicamente de dos tipos: aceleración (*speedup*) y tráfico de red generado. Como se ha establecido en la sección 5.6, el tiempo de ejecución para cada número de procesadores viene dado por el procesador más lento, mientras que el tráfico de red muestra el total de eventos generados por todos los nodos, lo que da una idea del comportamiento global de la aplicación.

6.3.1 Aceleración

La figura 6.1 muestra las curvas de *speedup* para los programas del repertorio Splash-2 descritos. En todos los casos puede verse que el rendimiento obtenido por el protocolo VSR-COMA a presiones de memoria del 25% son cercanos a los obtenidos por un sistema sin reemplazo. Exceptuando Radiosity, las aceleraciones obtenidas para una presión de memoria del 25% y 16 procesadores rondan el factor 8 en el resto de casos: desde el 7,46 de Ocean hasta el 8,17 de FFT, estando el resto de programas entre ambos valores: Barnes con 7,85, Radix con 8,02 y LU con 8,15. La gran variabilidad en el patrón de accesos en Radiosity penaliza su funcionamiento, con una aceleración de 2,75.

Al aumentar la presión de memoria cobra importancia la localidad espacial en el acceso a la memoria compartida de cada una de las aplicaciones. Por este motivo, con presiones del 50% se observa una mayor variabilidad en los resultados de aceleración, que van desde el 2,14 obtenido para LU hasta el 7,6 que se obtiene para FFT. A presiones del 80%, en la que se dispone de muy poco espacio disponible para replicación y que además obliga a realizar gran cantidad de operaciones de reemplazo, los resultados son aún más variables, ya que dependen en mucha mayor medida del tamaño del conjunto de trabajo que utiliza cada nodo. El mínimo lo presenta nuevamente LU, con un valor de 1,39, y el máximo es para FFT, con 7,10.

Los resultados obtenidos indican que el protocolo VSR-COMA y su es-

trategia de reemplazo asociada permiten una adecuada distribución del espacio de direcciones compartido entre las memorias de atracción, obteniéndose aceleraciones importantes en la ejecución de las cargas analizadas.

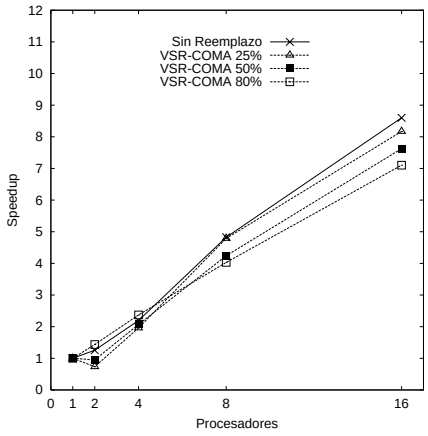
En algunas circunstancias, los resultados obtenidos por VSR-COMA a diferentes presiones son mejores que los de un sistema sin reemplazo. En concreto, esto sucede para los resultados de FFT en las ejecuciones con 2 y 4 nodos. El motivo es el siguiente. En la ejecución con los tamaños de AM iguales al tamaño del espacio de direcciones compartido, dicho espacio reside por completo en la memoria de atracción del nodo 0 al inicio de la ejecución de la aplicación. Esto hace que se produzcan un número importante de faltas frías en los primeros accesos de los nodos remotos a la memoria compartida. Cuando existe una presión de memoria mayor, el espacio de direcciones compartido no puede almacenarse en una única AM y en consecuencia durante la fase de inicialización el espacio de direcciones se distribuye entre los nodos. Esta distribución inicial, no computada ya que pertenece a la fase de inicialización de la aplicación, evita algunas de las faltas remotas que se producen en el caso del sistema sin reemplazo. El número de faltas así evitadas será pequeño, pero dado que cada falta remota supone un tiempo varios órdenes de magnitud mayor que en el caso de un acierto, este número de faltas puede influir en el resultado final cuando la tarea conlleva un número relativamente pequeño de accesos remotos, como es el caso de FFT. Al aumentar el número de procesadores, dicho efecto queda enmascarado por las faltas en caliente debidas tanto a conflicto y capacidad como a los accesos a las estructuras compartidas de sincronización.

6.3.2 Tráfico de red

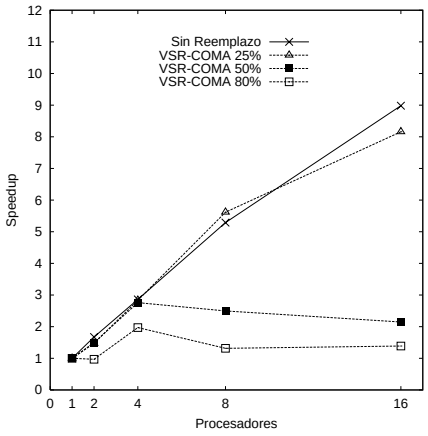
Las curvas de *speedup* de la figura 6.1 hacen referencia al cociente entre el tiempo consumido por la ejecución del programa con un solo procesador y el tiempo consumido por el más lento de los procesadores en el resto de ejecuciones. Por lo tanto, las curvas de *speedup* no permiten evaluar el funcionamiento general del sistema en cuanto a la eficiencia en el uso de los recursos, más concretamente de la red. Las gráficas de evolución del tráfico permiten comparar el tráfico generado a las diferentes presiones de memoria, lo que da una idea de la utilización de la red en la ejecución de los programas.

La figura 6.2 muestra la evolución del tráfico en VSR-COMA para las ejecuciones de los *kernels* FFT, LU y Radix, medido en número total de eventos intercambiados a través de la red. Dado que el número de eventos intercambiados varía según el número de nodos, para facilitar la comparación los resultados se han normalizado respecto a los eventos producidos en la ejecución de cada programa haciendo uso del sistema sin reemplazo.

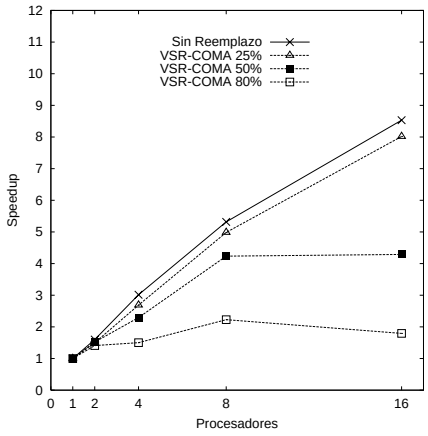
Como puede verse en la figura 6.2, el tráfico total para FFT a las diferen-



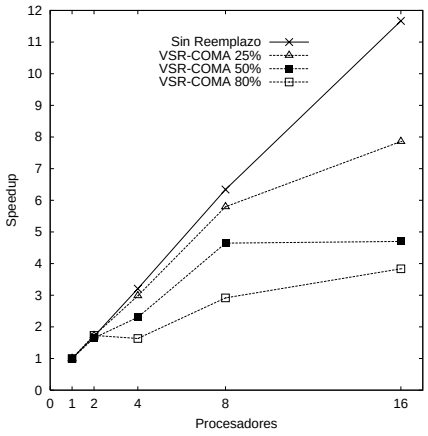
(a) FFT



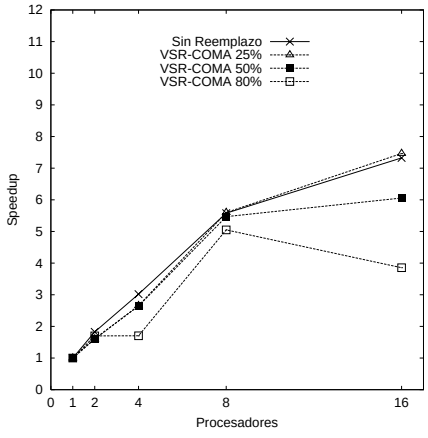
(b) Factorización LU



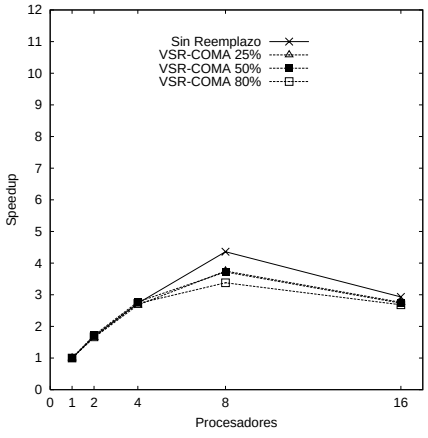
(c) Radix



(d) Barnes



(e) Ocean



(f) Radiosity

Figura 6.1: Comparativa de rendimientos de VSR-COMA a diferentes presiones de memoria.

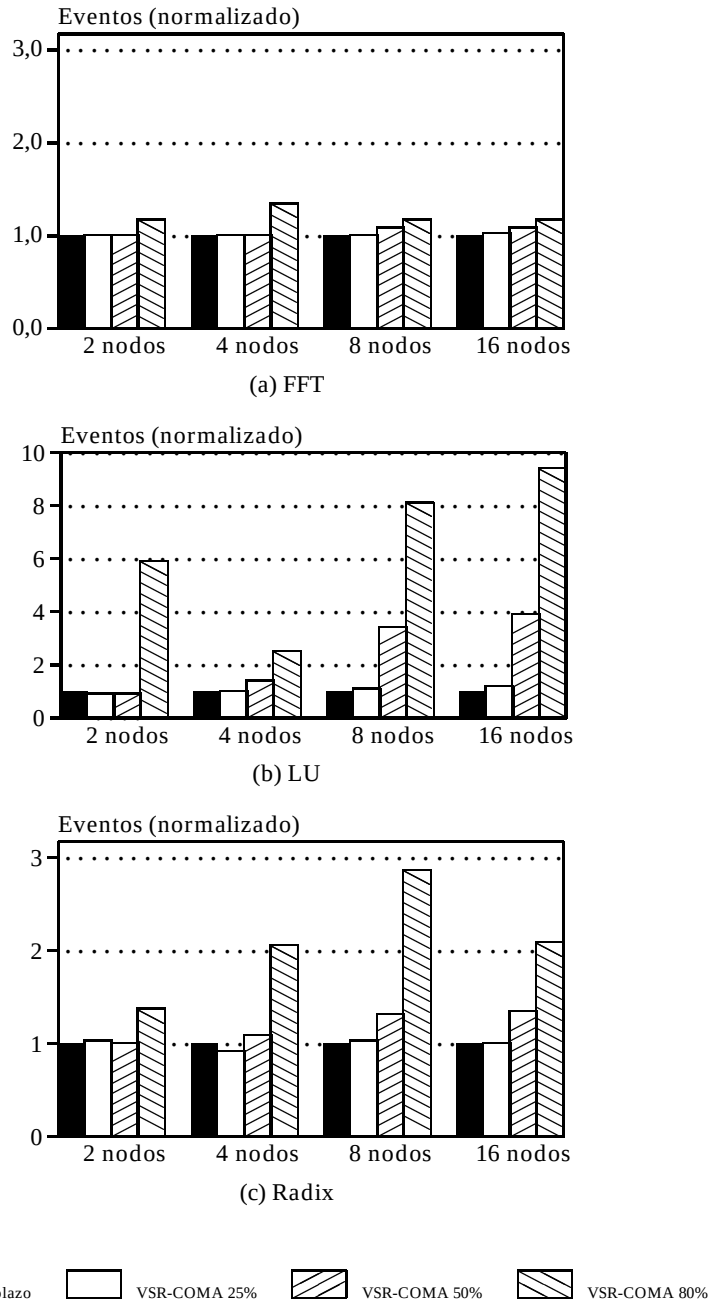


Figura 6.2: Evolución del tráfico para VSR-COMA a diferentes presiones de memoria y con una asociatividad de 8 vías. Los resultados están normalizados respecto a los eventos generados por el sistema sin reemplazo para cada número de nodos.

tes presiones de memoria es prácticamente el mismo que si no se produjeran faltas de capacidad o conflicto, mientras que para Radix y LU el tráfico crece para presiones altas. Concretamente, en LU el tráfico crece entre 2 y 4 veces para una presión del 50%, y llega a crecer hasta casi 10 veces para una presión del 80%. El crecimiento para Radix es mucho menor.

Algo similar sucede en la evolución del tráfico en la ejecución de las aplicaciones (figura 6.3), en donde Barnes genera un aumento de tráfico a presiones altas, mientras que el tráfico generado en Ocean y en Radiosity es prácticamente el mismo que si no existieran problemas de espacio en las memorias de atracción locales.

El crecimiento del tráfico a presiones altas para determinados programas depende en primer lugar de que cada nodo disponga de espacio suficiente en los conjuntos correspondientes de su memoria de atracción para almacenar los datos necesarios para la ejecución de su tarea. Si esto no se cumple, se genera un mayor número de faltas remotas y de solicitudes de reemplazo, con lo que el número total de eventos se incrementa de forma apreciable.

Sin embargo, la capacidad de las memorias de atracción locales no son el único factor del que depende el crecimiento del tráfico. Otro factor de importancia es la capacidad del protocolo de desalojar los bloques reemplazados hacia los nodos que tengan más posibilidades de utilizarlos, lo que disminuye la tasa de fallos en el futuro. Como hemos visto en la sección 4.2, uno de los principales objetivos de diseño del protocolo VSR-COMA es la utilización de estrategias de selección del nodo destino que minimicen el tráfico de red, basándose en el estado de las memorias de atracción remotas. El uso de esta estrategia ayuda a que el tráfico de red se mantenga prácticamente constante (como es el caso de FFT, Ocean o Radiosity) o crezca moderadamente (como sucede en LU, Radix o Barnes).

Para determinar la importancia de las estrategias de reemplazo en el funcionamiento del protocolo VSR-COMA, compararemos los resultados obtenidos a través de nuestra estrategia con los que se obtienen al utilizar otras estrategias de reemplazo descritas en la bibliografía.

6.4 Comparativa de estrategias de reemplazo

A continuación se presentarán los resultados de los experimentos realizados para comparar el funcionamiento de las diferentes estrategias de reemplazo en la ejecución de los programas del repertorio Splash-2. Se ha decidido la utilización de memorias de atracción asociativas por conjuntos de 4 vías, bloques de 256 bytes y una presión de memoria del 80%. Con estas condiciones de funcionamiento, la mejora en el rendimiento que permite la distribución de tareas entre los nodos se ve seriamente perjudicada por las faltas de capacidad y de conflicto. Es en estas circunstancias cuando cobra

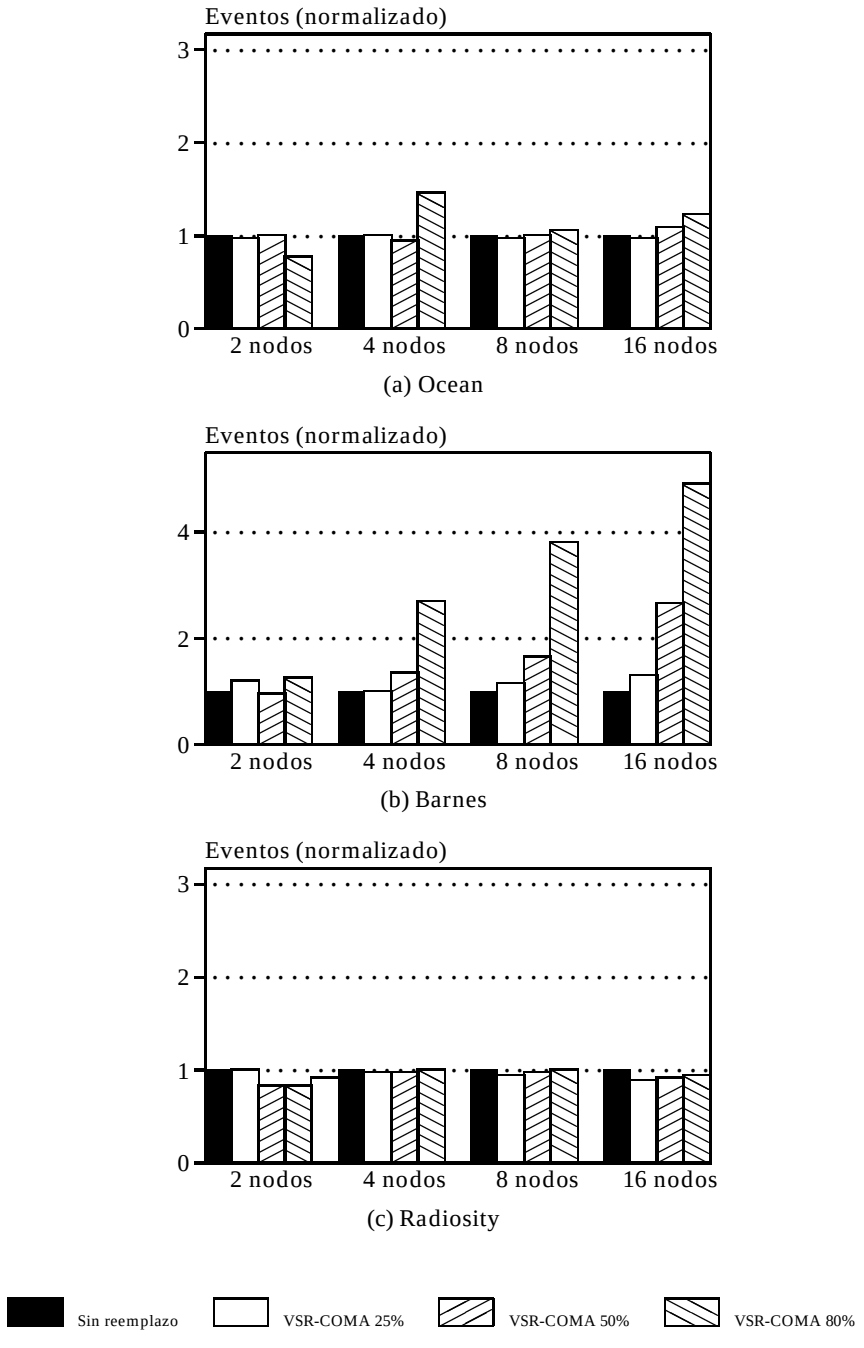


Figura 6.3: Evolución del tráfico para VSR-COMA a diferentes presiones de memoria y con una asociatividad de 8 vías. Los resultados están normalizados respecto a los eventos generados por el sistema sin reemplazo para cada número de nodos.

importancia la utilización de un mecanismo de reemplazo adecuado.

Como se ha descrito en el capítulo 4, una de las características del protocolo VSR-COMA es su independencia del mecanismo de selección del nodo destino utilizado por la técnica de reemplazo. Esto permite utilizar diferentes módulos de selección al objeto de comparar los rendimientos. Aprovechando esta característica, se han implementado tres módulos diferentes:

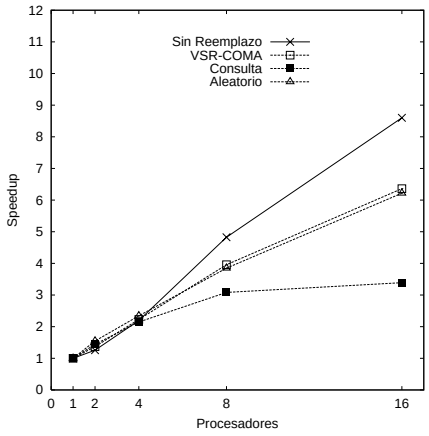
- Un módulo de selección aleatoria del nodo destino, con los mismos criterios que utiliza COMA-F, y que han sido descritos en la sección 3.3.1.
- Un módulo de selección por consulta del nodo destino, construido según las especificaciones de DICE, descritas en la sección 3.3.2.
- El módulo de selección de nodo destino basado en la estrategia de reemplazo de VSR-COMA, descrita en la sección 4.10.

La selección aleatoria de nodo destino no obliga a realizar ningún cambio en el funcionamiento del protocolo, ya que para realizarla sólo es necesario conocer el número total de nodos y la identidad del nodo local. Sin embargo, la utilización de la selección por consulta obliga a realizar algunos cambios menores en el protocolo VSR-COMA, al objeto de añadir los eventos de consulta y de respuesta (eventos BusAsk y BusAnsw, respectivamente), y tres nuevos eventos que permitan el intercambio de bloques: los eventos BusSreq (solicitud de intercambio de bloque), BusSack (respuesta positiva de intercambio de bloque) y BusSnak (respuesta negativa de intercambio de bloque). Estos tres últimos eventos posibilitan el intercambio de bloques cuando la presión de memoria es tan alta que todos los nodos responden a la consulta indicando que no pueden aceptar el bloque (ver sección 3.3.2). La utilización de estos nuevos eventos en el protocolo VSR-COMA, una vez verificado su correcto funcionamiento, ha permitido reproducir fielmente el funcionamiento del sistema de reemplazo por consulta.

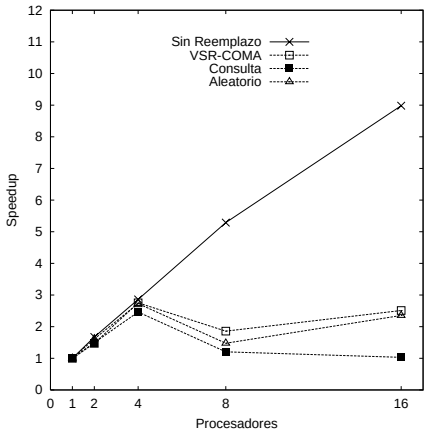
Los resultados obtenidos a través de la utilización de los tres mecanismos descritos se compararán con los que se obtienen utilizando un sistema VSR-COMA con espacio suficiente en cada memoria de atracción para almacenar el espacio compartido de direcciones de la aplicación. Esto supone la ausencia de mecanismos de reemplazo.

6.4.1 Aceleración

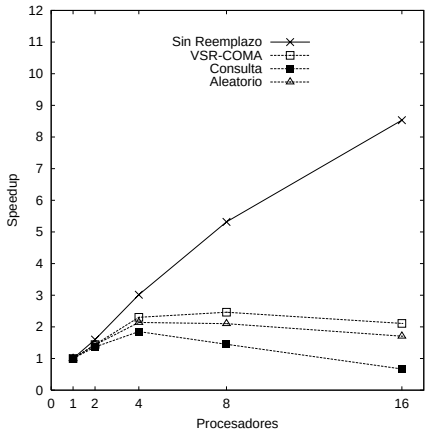
La figura 6.4 muestra las curvas de rendimiento obtenidas para las tres estrategias de reemplazo estudiadas, suponiendo la utilización de una red de interconexión tipo Myrinet.



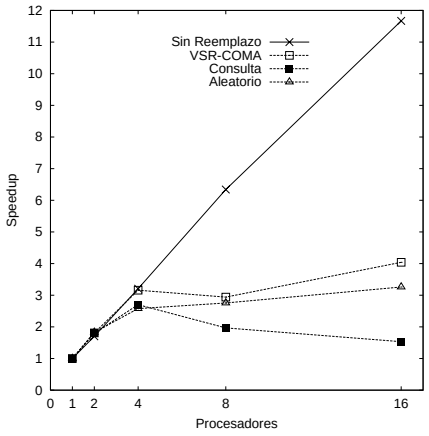
(a) FFT



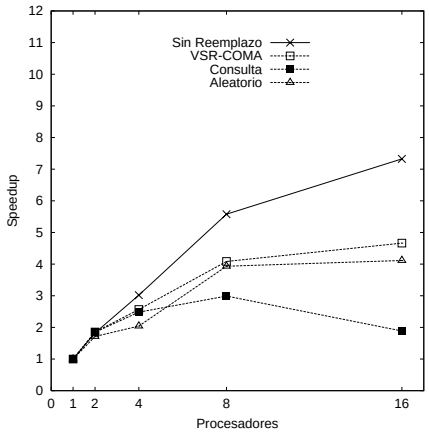
(b) Factorización LU



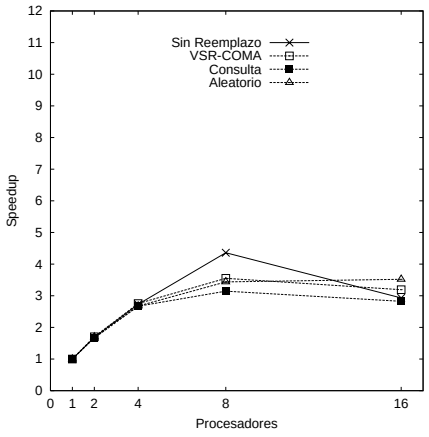
(c) Radix



(d) Barnes



(e) Ocean



(f) Radiosity

Figura 6.4: Comparativa de rendimientos para las tres estrategias de reemplazo estudiadas respecto a un sistema sin reemplazo.

Dado que la comparación se ha realizado a presiones de memoria altas, las diferencias en la aceleración obtenida no son muy grandes en términos absolutos, pero el hecho de utilizar el mismo protocolo de coherencia con diferentes estrategias de reemplazo nos permite asegurar que las diferencias que se observan se deben exclusivamente a dichas estrategias.

Los resultados obtenidos para el caso de 16 nodos muestra un incremento en la aceleración para la estrategia de VSR-COMA respecto de la estrategia de reemplazo por consulta con factores que van desde el 113% para el caso de Radiosity hasta el 315% para Radix. Los restantes programas presentan valores intermedios: 187% para FFT, 243% para LU, 247% para Ocean y 263% para Barnes.

Los incrementos en la aceleración obtenidos por la estrategia de reemplazo utilizada para VSR-COMA en relación con el mecanismo de reemplazo aleatorio son más modestos que para el reemplazo por consulta, y van desde el 102% respecto de FFT hasta el 124% que se obtienen tanto para Radix como para Barnes. Entre ambos valores se situan LU (106%) y Ocean (113%). En el caso de Radiosity, se da la circunstancia de que para 16 nodos el mecanismo de reemplazo aleatorio se comporta mejor que el mecanismo propuesto para VSR-COMA, arrojando un incremento con un factor de 90%, es decir, una reducción del 10% respecto a la aceleración obtenida mediante el reemplazo aleatorio.

A la vista de los resultados obtenidos puede apreciarse el bajo rendimiento de la estrategia de reemplazo basada en consulta. Este resultado era previsible si tenemos en cuenta que la selección basada en consulta, pese a seleccionar el nodo destino con mayores posibilidades de acierto, lo consigue a costa de generar un gran número de eventos en la red. Por otra parte, es de destacar el buen comportamiento de la selección aleatoria, que en todos los casos produce mejores resultados que la selección basada en consulta y aceptables en comparación con los producidos por VSR-COMA. En el caso de FFT, los resultados son prácticamente iguales, mientras que para Radiosity llega a superar a VSR-COMA en la ejecución con 16 nodos.

En el resto de los casos puede observarse que la aceleración obtenida para la estrategia de reemplazo de VSR-COMA es mayor que la obtenida por el resto de estrategias, lo que demuestra la validez de la estrategia de reemplazo propuesta en este trabajo.

6.4.2 Tráfico de red

El estudio del tráfico de red nos permite observar el comportamiento global de las diferentes estrategias de reemplazo para cada uno de los programas ejecutados. La figura 6.5 muestra la evolución del tráfico para los *kernels* FFT, LU y Radix utilizando los mecanismos de reemplazo descritos más arriba. Como puede apreciarse, el número total de eventos generado por el mecanismo de reemplazo por consulta es mucho mayor que los gene-

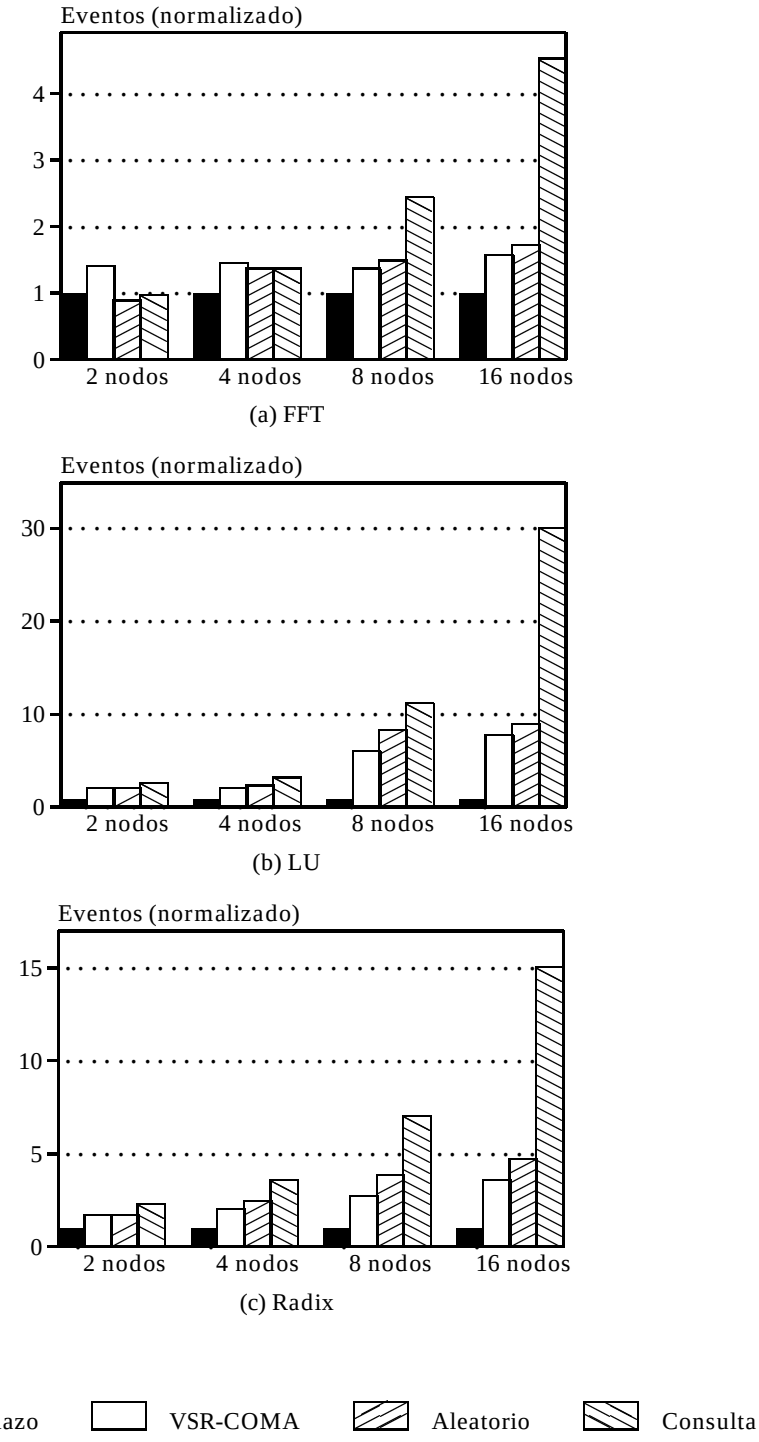


Figura 6.5: Evolución del tráfico para las diferentes estrategias de reemplazo. Los resultados están normalizados respecto a los eventos generados por el sistema sin reemplazo para cada número de nodos.

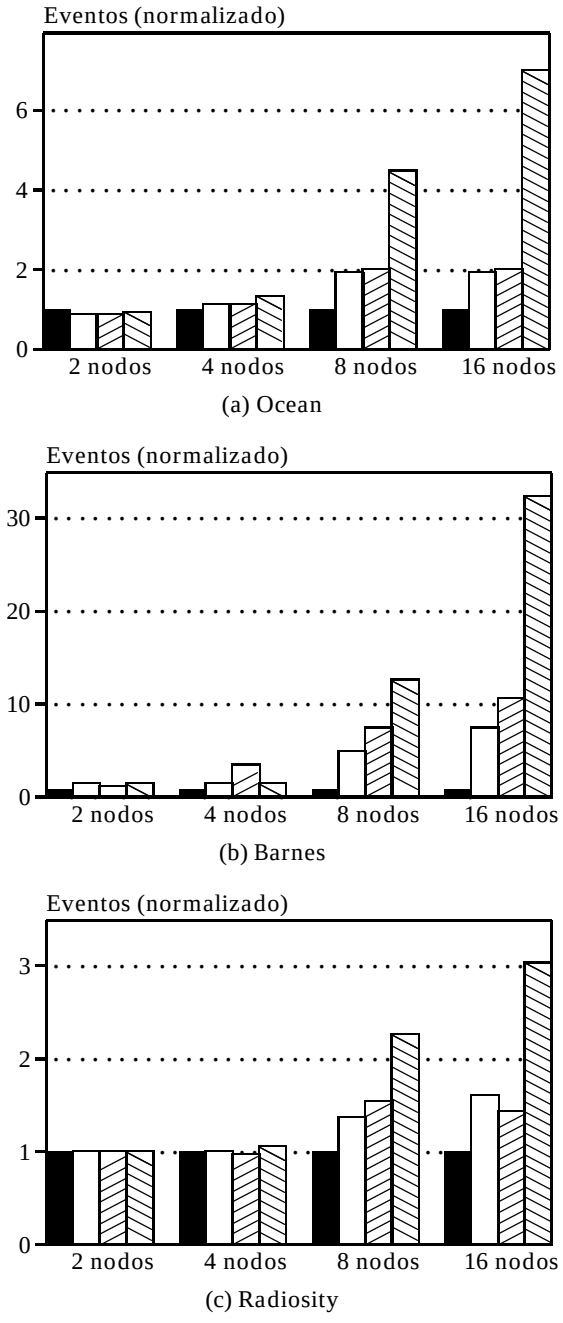


Figura 6.6: Evolución del tráfico para las diferentes estrategias de reemplazo. Los resultados están normalizados respecto a los eventos generados por el sistema sin reemplazo para cada número de nodos.

rados por los restantes mecanismos, sobre todo cuando crece el número de nodos. Respecto a la estrategia de reemplazo aleatorio, el número de eventos generados es similar a los generados utilizando la estrategia de VSR-COMA. Para FFT, la proporción de eventos generados por la estrategia de VSR-COMA es prácticamente constante respecto de la generada por un sistema sin reemplazo, mientras que la proporción de eventos generados por el resto de estrategias crece con el número de nodos. Para LU y Radix puede apreciarse que la estrategia de reemplazo basada en consulta genera un número de eventos 30 y 15 veces mayor, respectivamente, que una estrategia sin reemplazo, mientras que la estrategia de VSR-COMA genera un número mucho menor de eventos y la estrategia aleatoria muestra un comportamiento aceptable.

La figura 6.6 muestra el funcionamiento de las estrategias de reemplazo descritas en la ejecución de las aplicaciones. De nuevo se aprecian diferencias importantes en el número de eventos generados por el mecanismo de reemplazo por consulta respecto de las demás estrategias. Al igual que en la ejecución de los *kernels*, el número total de eventos generados por el reemplazo aleatorio es comparable con los generados por el algoritmo de reemplazo de VSR-COMA. Sin embargo, los criterios de selección de nodo destino utilizados por VSR-COMA permiten mejorar el rendimiento. En el caso de Ocean, ambos criterios generan un número de eventos de red muy similar, pese a lo cual la selección de nodo destino que realiza VSR-COMA mejora claramente la aceleración obtenida.

6.5 La cuestión de la sobrecarga de memoria

Como se ha visto en las secciones anteriores, el protocolo VSR-COMA permite acelerar la ejecución de los programas y disminuir el tráfico de red generado. Para conseguirlo, VSR-COMA utiliza un mecanismo de reemplazo basado en el mantenimiento en cada controlador de coherencia de información específica. Como se ha descrito en la sección 4.3, dicha información consiste en el estado y la etiqueta asociadas a todos los marcos de bloque de cada uno de los nodos remotos. Queda por responder una pregunta importante: si el mantenimiento de dicha información supone un incremento excesivo de la memoria de control asociada a cada uno de los controladores de coherencia.

La respuesta es negativa. En la tabla 6.2 se recogen los datos relativos a los tamaños de la información de control que debe mantener cada controlador de coherencia para ejecutar los programas del repertorio Splash-2 utilizados. Los datos se han calculado en base a las condiciones de ejecución utilizadas para la comparativa de rendimientos de la sección 6.4: presión de memoria del 80%, bloques de 256 bytes y 4 vías por conjunto. La columna AM hace referencia al tamaño en bytes de cada una de las memorias de

Aplicación	Nodos	AM (bytes)	Estado + Tag (bytes)	Directorio (bytes)	Reemplazo (bytes)	Control (E+D+R) (bytes)	Sobrecarga VSR-COMA (%)	Sobrecarga resto (%)
FFT	2	2113536	5160	1650	5160	11970	0,56	0,32
FFT	4	1056768	3096	3300	9288	15684	1,48	0,60
FFT	8	528384	1806	4950	12642	19398	3,67	1,27
FFT	16	266240	1040	6600	15600	23240	8,72	2,86
LU	2	409600	1200	320	1200	2720	0,66	0,37
LU	4	204800	700	640	2100	3440	1,67	0,65
LU	8	102400	400	960	2800	4160	4,06	1,32
LU	16	53248	234	1280	3510	5024	9,43	2,84
Radix	2	5898240	17280	4608	17280	39168	0,66	0,37
Radix	4	2949120	10080	9216	30240	49536	1,67	0,65
Radix	8	1474560	5760	13824	40320	59904	4,06	1,32
Radix	16	737280	3240	18432	48600	70272	9,53	2,93
Ocean	2	10485760	25600	8192	25600	59392	0,56	0,32
Ocean	4	5242880	15360	16384	46080	77824	1,48	0,60
Ocean	8	2621440	8960	24576	62720	96256	3,67	1,27
Ocean	16	1310720	5120	32768	76800	114688	8,75	2,89
Barnes	2	3276800	9600	2560	9600	21760	0,66	0,37
Barnes	4	1638400	5600	5120	16800	27520	1,67	0,65
Barnes	8	819200	3200	7680	22400	33280	4,06	1,32
Barnes	16	409600	1800	10240	27000	39040	9,53	2,93
Radiosity	2	20971520	51200	16384	51200	118784	0,56	0,32
Radiosity	4	10485760	30720	32768	92160	155648	1,48	0,60
Radiosity	8	5242880	17920	49152	125440	192512	3,67	1,27
Radiosity	16	2621440	10240	65536	153600	229376	8,75	2,89

Tabla 6.2: Sobrecarga debida a la memoria de control en las ejecuciones de los programas del repertorio Splash-2, para VSR-COMA y para el resto de estrategias de reemplazo analizadas.

atracción: dicho tamaño cambia en función del número de nodos, al objeto de mantener la presión de memoria constante. La columna "Estado + Tag" muestra el tamaño correspondiente de la información de estado y etiqueta que deberá mantener cada controlador de coherencia. Lo mismo se aplica a las columnas "Directorio" y "Reemplazo". La columna "Control", por su parte, contiene la suma de las tres columnas anteriores.

La columna "Sobrecarga VSR-COMA" muestra en términos porcentuales la relación entre la información de control presente en cada controlador de coherencia del sistema VSR-COMA y el tamaño de la AM utilizado en cada caso. Finalmente, la columna "Sobrecarga resto" muestra la sobrecarga de memoria debida al uso de información de estado y etiqueta e información de directorio, esto es, la sobrecarga si no tenemos en cuenta la información de reemplazo. Esta sobrecarga es la asociada a la utilización de VSR-COMA con los restantes mecanismos de reemplazo examinados: aleatorio, por consulta o un sistema sin reemplazo. La sobrecarga de memoria será menor en estos casos, ya que no se mantiene información de reemplazo.

Los resultados indican que el mantenimiento de la información de reemplazo supone un incremento de la información de control que debe mantenerse en cada controlador de coherencia. Dicho supone almacenar aproximadamente entre 2 y 4 veces más información que si sólo se almacenara el estado de la memoria de control local y la información de directorio, llegando en algunos casos a representar casi el 10% del tamaño de la memoria de atracción.

El incremento en la sobrecarga de memoria debida al mantenimiento de la información de reemplazo es la principal desventaja que presenta la estrategia de reemplazo producida por el protocolo VSR-COMA en comparación con las otras estrategias estudiadas. A cambio de este incremento, la estrategia de VSR-COMA permite la obtención de mejores aceleraciones y una mayor eficiencia en el uso de la red.

6.6 Conclusiones

En este capítulo se ha estudiado el rendimiento del protocolo VSR-COMA a diferentes presiones de memoria, tanto en lo que respecta a la aceleración en la ejecución de programas pertenecientes al protocolo Splash-2 como al tráfico de red generado. Los resultados obtenidos presentan valores de aceleración para 16 nodos de hasta 8,17 para presiones de memoria del 25%, de hasta 7,6 para presiones del 50% y de hasta 7,10 para presiones del 80%. Estos resultados muestran que la estrategia de reemplazo utilizada por VSR-COMA para distribuir el espacio de direcciones compartido entre las memorias de atracción es acertada, ya que se obtienen buenos tiempos de ejecución de los programas paralelos utilizados en la evaluación de ren-

dimientos, sobre todo a presiones de memoria bajas.

Para comparar el funcionamiento de la estrategia de reemplazo propuesta en VSR-COMA con otras estrategias propuestas en la bibliografía se ha realizado un estudio comparativo a presiones de memoria altas, utilizando el protocolo VSR-COMA como soporte e implementando sobre él las diferentes estrategias de reemplazo. Los resultados muestran que la estrategia de VSR-COMA permite obtener mejores aceleraciones que el resto de estrategias de reemplazo, con incrementos de hasta un 315% para la estrategia basada en consulta y de hasta un 124% respecto de la estrategia de reemplazo aleatorio, disminuyéndose además el número de eventos transmitidos a través de la red, uno de los objetivos principales del diseño de VSR-COMA.

Las mejoras en el rendimiento para la estrategia de reemplazo de VSR-COMA se consiguen a través del mantenimiento de la información de reemplazo en los controladores de coherencia de cada nodo. Esto produce una cierta sobrecarga en la información de control que debe mantener cada controlador de coherencia. Dicha sobrecarga, que ha sido cuantificada para las cargas de trabajo ejecutadas, puede llegar en algunos casos hasta casi el 10% del tamaño de la memoria de atracción, y constituye la principal desventaja que presenta la utilización de la estrategia de reemplazo de VSR-COMA. A cambio, se ha comprobado que dicha estrategia aumenta significativamente la aceleración en la ejecución de aplicaciones paralelas.

Capítulo 7

Conclusiones

Se ha realizado un estudio sobre los sistemas de memoria compartida distribuida, clasificándolos en función de la localización y almacenamiento de los bloques que componen la memoria compartida direccionable. Se ha prestado una especial atención a la arquitectura COMA, describiendo su funcionamiento general y su mecanismo de localización y reemplazo de bloques.

Se ha realizado un estudio en profundidad del problema del reemplazo en las arquitecturas COMA. Se ha descrito el problema en detalle y se han examinado las soluciones propuestas en la bibliografía, estudiando los protocolos de coherencia que las implementan y analizando sus ventajas e inconvenientes. Se ha propuesto además un nuevo mecanismo de reemplazo, basado en el mantenimiento local de la información de estado de las memorias de atracción remotas. Dicho mecanismo permite seleccionar el nodo destino de una operación de reemplazo minimizando el tráfico generado en la red de interconexión.

Se presenta un nuevo protocolo, VSR-COMA, apto para su funcionamiento en un entorno débilmente acoplado formado por un conjunto de estaciones de trabajo y una red de interconexión de tipo bus. VSR-COMA permite gestionar la compartición de memoria según el planteamiento de una arquitectura COMA. El protocolo VSR-COMA se ha verificado utilizando redes de Petri coloreadas, lo que ha permitido eliminar errores de diseño y verificar su funcionamiento en las primeras etapas de su desarrollo. Se ha descrito tanto el protocolo de memoria de VSR-COMA, que permite a cada procesador la realización de operaciones sobre el espacio compartido de direcciones, como el protocolo de coherencia, que posibilita a cada controlador el mantenimiento de la coherencia en su memoria de atracción local. Asimismo, se ha descrito en profundidad la gestión de eventos entrantes que realiza VSR-COMA, prestando especial atención a la posibilidad del solapamiento de transacciones y a las condiciones de carrera que dicha situación pueda generar. La independencia entre el protocolo

y el mecanismo de reemplazo de bloques en VSR-COMA posibilita la construcción de diferentes módulos de selección de nodos destino en operaciones de reemplazo.

Se ha construido un simulador, llamado EMUCOMA, que permite simular la ejecución de aplicaciones paralelas en un sistema multicomputador que utiliza el protocolo VSR-COMA para el mantenimiento de la coherencia entre las diferentes memorias de atracción. Se ha descrito además un modelo de programación que permite el desarrollo de aplicaciones paralelas para su ejecución con EMUCOMA que es independiente de las características de la arquitectura subyacente, utilizando para ello las macros Parmacs.

Se ha desarrollado un modelo analítico que permite la comparación directa de la velocidad de ejecución de diferentes aplicaciones, a través de la utilización de los índices de rendimiento generados por EMUCOMA para calcular la aceleración de las aplicaciones al ser ejecutadas en el sistema multicomputador simulado.

Se ha utilizado un conjunto de programas pertenecientes al repertorio Splash-2 para medir el rendimiento del sistema VSR-COMA, utilizando el simulador EMUCOMA y el modelo analítico desarrollado.

Se ha estudiado el rendimiento de un sistema multicomputador con el protocolo VSR-COMA en la ejecución de las aplicaciones para diferentes presiones de memoria, tanto en lo que respecta a la mejora en la velocidad de ejecución como en lo referido al tráfico total de red generado en cada caso. Los resultados de aceleración obtenidos para 16 nodos muestran una aceleración de hasta 8,17 para presiones de memoria del 25%, de hasta 7,6 para presiones del 50% y de hasta 7,10 para presiones del 80%.

El rendimiento del mecanismo de selección de nodo destino propuesto en VSR-COMA se ha comparado con los mecanismos de selección aleatoria y selección por consulta propuestos en la bibliografía, para el caso de un sistema con una presión de memoria del 80%. Los resultados se han comparado con los que se obtendrían con un sistema sin reemplazo, y se presentan en términos de aceleración y de tráfico total de red para cada uno de las estrategias de reemplazo examinadas. Los resultados obtenidos muestran que VSR-COMA permite obtener un incremento en la aceleración de hasta un 315% con respecto al mecanismo de selección por consulta y de hasta un 124% con respecto al mecanismo de selección aleatoria, reduciendo además el número de eventos necesarios para el mantenimiento de la coherencia.

Finalmente, se ha evaluado el coste en términos de sobrecarga de memoria asociado al mantenimiento de la información de reemplazo en el protocolo VSR-COMA para cada una de las aplicaciones utilizadas. Se ha comparado dicha sobrecarga con la necesaria para el funcionamiento del protocolo con otras estrategias de reemplazo. Los resultados han permitido observar que VSR-COMA genera una sobrecarga que puede llegar a

representar hasta el 10% del tamaño de la memoria de atracción, y que es entre 2 y 4 veces mayor que la requerida por el resto de mecanismos de reemplazo.

Apéndices

Apéndice A

Modelado de VSR-COMA con redes de Petri coloreadas

A.1 Introducción

Una manera rápida de determinar el funcionamiento de un sistema a través de su definición detallada es el modelado. El protocolo VSR-COMA ha sido modelado utilizando redes de Petri coloreadas, una técnica que permite construir un modelo de procesos estocásticos al objeto de modelar su comportamiento. En las siguientes secciones haremos una breve introducción a las redes de Petri y detallaremos la estructura y funcionamiento del modelo construido.

A.2 Las redes de Petri

Una red de Petri [78] se compone de un grafo dirigido bipartito y un estado inicial, llamado *marcado inicial*. Las dos particiones se denominan *lugares* y *transiciones*. Desde un punto de vista gráfico, los lugares se dibujan como círculos y las transiciones como rectángulos. Los arcos del grafo tienen asociado un *peso*, esto es, una etiqueta con un número entero positivo, y van desde un lugar a una transición o viceversa. Un arco con un peso k puede interpretarse como k arcos paralelos con un peso de 1. Si un arco no posee etiqueta se le supone un peso de 1.

El estado del sistema modelado por la red se representa a través de la asignación de números naturales a los lugares. Este procedimiento se denomina *marcado*. Para facilitar la representación gráfica, en lugar de utilizar números enteros se utilizan puntos negros, denominados *fichas* o *tokens*, que se colocan dentro de los lugares. Cada token tiene un valor de 1. Típicamente, los lugares representan algún tipo de estado o condición mientras que las transiciones representan eventos que suceden en el sistema. De esta manera, los lugares de entrada y de salida de una transición representan

pre y post-condiciones.

A.2.1 Habilitación y disparo de transiciones

Los cambios de estado del sistema se modelan a través de las reglas de habilitación y disparo. Dichas reglas pueden describirse como sigue:

1. Una transición t está habilitada bajo un marcado M si cada lugar de entrada p de t posee un marcado de al menos un número de tokens igual al peso del arco de entrada a la transición a partir de dicho lugar p.

2. Una transición puede dispararse o no al estar habilitada. Cuando más de una transición está habilitada, la que se dispara en primer lugar se selecciona de una forma no determinista. El disparo de una transición representa la ocurrencia de un evento en el sistema.

3. El disparo de una transición conlleva la retirada de un número de tokens de cada lugar de entrada igual al peso del arco de entrada que conecta dicho lugar con la transición, y el añadido de un número de tokens igual al peso del arco de salida en cada uno de los lugares de salida de la transición.

Una transición puede tener cero arcos de entrada (transición fuente o productora) o cero arcos de salida (transición consumidora). Por lo tanto, a medida que las transiciones se van disparando, el número total de tokens presentes en todos los lugares del sistema puede ir variando.

La figura A.1 muestra un ejemplo de disparo de transiciones. Partiendo del estado inicial (A), la red tiene dos posibilidades de evolución diferentes, ya que ambas transiciones están habilitadas. Si la primera transición en dispararse es t1, se llega al estado B, en donde no quedan transiciones habilitadas. Si la primera en dispararse es t2, se pasa a la situación mostrada en (C), en la que la transición t2 continúa estando habilitada, al estar presentes los tokens de entrada necesarios. En consecuencia, se vuelve a disparar, llegando al estado final indicado en D.

A.2.2 Las redes de Petri coloreadas

En contraste con las redes de Petri descritas, en las que los tokens son de un único tipo, las "redes de Petri coloreadas" (CPN) [42, 43, 44] utilizan tokens que transportan un valor de un *tipo* determinado. De esta manera, los tokens pueden distinguirse unos de otros. Cada uno de los tipos de datos se denomina "conjunto de colores", mientras que el dato que transporta un token es su "color".

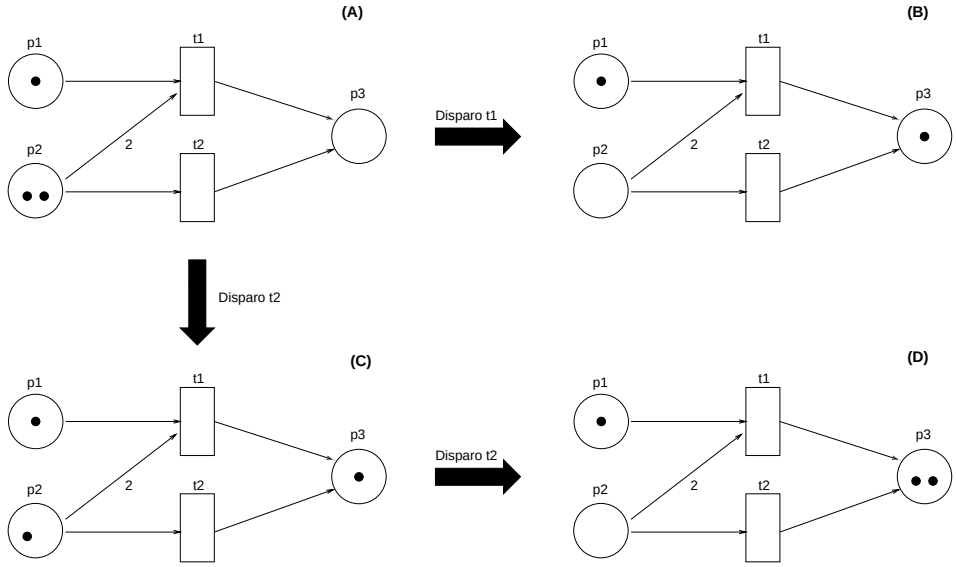


Figura A.1: Ejemplo de disparo de transiciones con redes de Petri.

Las reglas de disparo de las redes de Petri coloreadas son similares a las vistas en el apartado A.2.1. Para que una transición esté habilitada, debe disponer de un número suficiente de tokens en sus lugares de entrada. Además, los valores de esos tokens deben *corresponder* a las expresiones de los arcos de entrada.

La figura A.2 muestra un ejemplo de red de Petri coloreada. Como puede observarse, aparecen algunos elementos nuevos respecto a la figura A.1. En primer lugar, cada uno de los lugares tiene un tipo determinado. Dicho tipo puede ser un tipo básico o bien un producto de tipos. Por otra parte, cada dato o color presente en un lugar lleva una indicación de su *multiplicidad*, esto es, de cuántas copias del dato están presentes en el lugar. En nuestro ejemplo, el lugar p1 tiene cinco tokens: uno de ellos es la pareja (0,“cadena A”) y los cuatro restantes tienen el valor (18,“cadena B”).

En lo que respecta a los arcos de entrada y salida a las transiciones, pueden utilizarse variables para indicar datos genéricos (como se hace en el arco de entrada de p1 a t1) o bien datos (colores) concretos. En el primer caso cualquier dato será válido, mientras que en el segundo la transición sólo se habilitará cuando haya un dato de ese tipo en el lugar de entrada correspondiente. Cuando se dispare la transición, los datos colocados en los lugares de salida y su multiplicidad dependerán de las expresiones de los arcos de salida. En caso de utilizar variables en dichas expresiones, sus nombres deberán corresponder con las utilizadas en los arcos de entrada a la transición.

Las transiciones pueden tener o no una condición de *guarda* asociada a

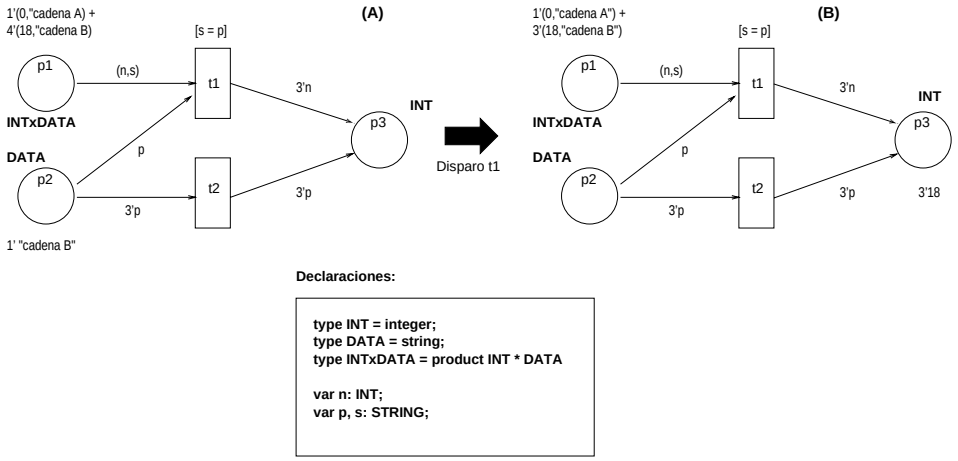


Figura A.2: Ejemplo de disparo de transiciones con redes de Petri coloreadas.

ellas. Si no la tienen, la transición se disparará cuando los tokens presentes en los lugares de entrada satisfagan las expresiones de los arcos de entrada. En caso de que exista una condición de guarda, la transición sólo se disparará si, además de existir los tokens correspondientes en los lugares de entrada, la condición se cumple.

En la red mostrada en la figura A.2, la transición t1 se habilita al existir en los lugares de entrada los tokens que cumplen la condición de guarda asociada a dicha transición. Una vez disparada ésta, se coloca el dato o color en el lugar de salida, con la multiplicidad indicada en el arco de salida.

A.2.3 Design/CPN

Design/CPN [16, 43] es una herramienta que permite la construcción y la ejecución de una red de Petri coloreada a partir de un marcado inicial. Entre sus principales características figura la posibilidad de utilizar una jerarquía de páginas, de modo que una transición sea en realidad una página con otra red de Petri. Como veremos en las secciones siguientes, el uso de estas *transiciones de substitución* facilita en gran medida el desarrollo de modelos complejos, al ser posible utilizar técnicas de diseño descendente.

A.3 Modelado de VSR-COMA

Las redes de Petri son un mecanismo muy utilizado en la construcción de modelos de sistemas con un comportamiento estocástico. Aunque los protocolos de coherencia COMA admiten la aplicación de otros métodos de

verificación [24], ciertos tipos de redes de Petri han demostrado su utilidad en el modelado de sistemas COMA y de sistemas de coherencia cache en entornos multiprocesador, como las redes de Petri temporizadas, que permiten la evaluación de rendimientos [100, 108] y las redes de Petri coloreadas, que facilitan la construcción de modelos de sistemas con un gran nivel de detalle [84, 10]. El componente estocástico de las redes de Petri facilita la generación de casos de prueba correspondientes a situaciones poco frecuentes, que pueden llevar a errores difíciles de detectar en las etapas iniciales del diseño [38]. Esta característica de las redes de Petri las hace especialmente apropiadas para el modelado de sistemas con interacciones estocásticas, como es el caso de los controladores de coherencia de sistemas COMA.

A continuación expondremos la estructura y funcionamiento del modelo de VSR-COMA a partir de redes de Petri coloreadas. Para ello, en primer lugar se verán los conjuntos de colores definidos, para luego seguir con la estructura jerárquica de las páginas que componen el modelo y la descripción de todas sus redes.

A.3.1 Tipos de datos

Los tipos de datos -"conjuntos de colores" en la terminología CPN- que se utilizan en el modelo son los que pueden verse en la figura A.3. Se han utilizado nombres ligeramente diferentes para los eventos de protocolo y los estados de los bloques del modelo a los utilizados en la descripción de VSR-COMA en el capítulo 4. Pese a ello, el funcionamiento del protocolo modelado no varía respecto al ya descrito para VSR-COMA. Las equivalencias entre los nombres aparecen como comentarios en la figura A.3.

Las declaraciones están divididas en tres partes: conjuntos de colores formados a partir de tipos básicos, conjuntos de colores compuestos y variables. El significado de los diferentes conjuntos de colores se irá aclarando a medida que nos adentremos en la descripción del modelo.

Como puede verse en las declaraciones, sólo se han definido dos operaciones de memoria: lectura (PrRd) y escritura (PrWr). Sin embargo, VSR-COMA incorpora un total de cuatro operaciones diferentes: PrRd, PrWr, PrTAS (*Test & Set*) y PrFAI (*Fetch & Increment*). Estas dos últimas operaciones no se han incluido en el modelo ya que esencialmente suponen una escritura sobre el bloque, aunque precedidas de una lectura. Como ambas acciones forman parte de la actualización de la información del bloque, pueden considerarse como escrituras a efectos del mantenimiento de la coherencia, por lo que el modelo basado en lecturas y escrituras es válido también para ellas.

```
(* --------Conjuntos de colores basicos ----------- *)

color ETIQUETA = with nulo | uno | dos | tres | cuatro | cinco
                   | seis | siete | ocho | nueve | diez
                   | once ;

(* Etiquetas de bloques. la simulacion utiliza tres nodos con cuatro
vias como maximo y la presion de memoria sera menor que el 100 por
100. Por lo tanto,  nos hacen falta 12-1=11 etiquetas diferentes *)

color TESTIGO = with ficha | noficha; (* Testigo para operaciones
           secuenciales *)

(* Estados posibles para cada marco de bloque. El estado SharOwn
(compartido propietario) se representa utilizando el estado Com
y la informacion de directorio. *)

color ESTADO = with   Inv  (* Invalid *)
     | Excl (* Exclusive *)
     | Com  (* Shared *)
     | I_E  (* InvWExcl *)
     | I_C  (* InvWShar *)
     | E_D  (* WExport *);

(* Numero de marco. Se numeran a partir de 0. El -1 codifica marco
invalido (para aquellos eventos que no lo necesiten *)
color NUM_MARCO = int;

color NODO = with A | B | C;
color CONTADOR = int;
color TIPOEVENTO = with SL (* BusRreq *)
     | RL   (* BusRack *)
                   | SE   (* BusWreq *)
                   | RE   (* BusWack *)
                   | IE   (* BusFInv *)
                   | RCC  (* BusRACE *)
                   | SD   (* BusEXreq *)
                   | RD   (* BusEXack *)
                   | RND  (* BusEXnak *);

(* Operaciones de memoria *)
color SOL_OPERACION = with Read | Write (* PrRd, PrWr *)
color RES_OPERACION = with ReadACK | WriteACK; (* Respuestas de operacion *)

(* --------Conjunto de colores compuestos de datos ------------------- *)

(* Operacion de memoria *)
color OPERACION = product SOL_OPERACION * ETIQUETA; (* Operacion de memoria *)

(* Contenido de un marco de bloque *)
color MARCO = product ETIQUETA * ESTADO * NUM_MARCO;

color DIRECTORIO = product ETIQUETA * NODO; (* Entradas de directorio *)

(* Inf. de reemplazo *)
color REEMPLAZO = product NODO * ETIQUETA * ESTADO * NUM_MARCO;

(* Evento de protocolo. En el modelo, solo se transmiten las
etiquetas. Si un evento no requiere numero de marco, aqui
codificaremos 99: en la practica ese evento no llevaria dicho numero. *)
color EVENTO= product TIPOEVENTO * NODO * ETIQUETA * NUM_MARCO * NODO;

(* Lista de eventos en red (= evento + numero de orden de llegada) *)
color EVENTO_EN_RED = product EVENTO * CONTADOR;

(* ------------------- Variables ----------------- *)

var etiqueta, etiq_am, etiq_dir, etiq_reemp: ETIQUETA;
var etiq_am0, etiq_am1, etiq_am2, etiq_am3: ETIQUETA;
var etiq_dir0, etiq_dir1, etiq_dir2, etiq_dir3: ETIQUETA;
var etiq_reemp1, etiq_reemp2, etiq_reemp3, etiq_reemp0: ETIQUETA;
var etiq_carrera: ETIQUETA;

var num_marco, num_marco_ori, num_marco_prop, num_marco2,
num_marco_reemp1, num_marco_reemp2, num_marco_reemp3, num_marco_reemp0,
num_marco_reemp, num_marco_am: NUM_MARCO;

var marco: MARCO;

var sol_operacion: SOL_OPERACION;
var res_operacion: RES_OPERACION;
var operacion: OPERACION;

var testigo: TESTIGO;
var directorio: DIRECTORIO;
var reemplazo: REEMPLAZO;

var estado, estado_am, estado_reemp : ESTADO;
var estado_am0,estado_am1, estado_am2, estado_am3: ESTADO;
var estado_reemp0, estado_reemp1, estado_reemp2, estado_reemp3: ESTADO;

(* Todos los tipos de nodos que pueden aparecer *)
var nodo_local, nodo_copia, nodo_prop, nodo_anfit, nodo_dir,
 nodo_reemp, nodo_ori, nodo_des, nodo,
 nodo_reemp1, nodo_reemp2, nodo_reemp3, nodo_reemp0: NODO;

var nodo_dir0, nodo_dir1, nodo_dir2, nodo_dir3: NODO;

var tipoevento: TIPOEVENTO;
var evento: EVENTO;
var evento_en_red: EVENTO_EN_RED;

var contador, contador2,orden: CONTADOR;
```

Figura A.3: Declaraciones de conjuntos de colores en el modelo de VSR-COMA.

A.3.2 Estructura del modelo

La figura A.4 muestra la estructura de páginas del modelo de VSR-COMA. El modelo realizado de VSR-COMA consta de tres nodos de proceso, denominados “NodoA”, “NodoB” y “NodoC”. Debido a que los tres nodos tienen la misma estructura de páginas, la figura A.4 sólo muestra desplegada la estructura para uno de ellos, “NodoA”.

La página principal es la de Sistema, que se encuentra en la parte superior de la jerarquía. Esta página contiene la descripción de la situación inicial en todos los nodos de proceso, así como en la red de interconexión (figura A.5). En el modelo de VSR-COMA desarrollado se ha establecido la simulación para un único conjunto de atracción de cuatro vías. El hecho de utilizar caches asociativas es indispensable para comprobar el funcionamiento del protocolo, sobre todo en lo que respecta al mecanismo de reemplazo de bloques.

A.3.3 Condiciones iniciales

El modelo propuesto tiene por cada nodo de proceso seis lugares básicos (figura A.5). Los lugares son los siguientes:

Identidad del nodo X : este lugar almacena un valor que indica la identidad del nodo.

“Pool” de operaciones : este lugar almacena las operaciones de memoria que van a realizarse (Read o Write). La figura muestra que la situación inicial de este lugar es de 50 operaciones de cada tipo.

“Pool” de etiquetas : almacena las etiquetas a las que va a accederse a través de la simulación. El número de etiquetas diferentes depende de la presión de memoria que se decida utilizar en la simulación. En el caso de la figura A.5 se dispone de diez etiquetas diferentes. Teniendo en cuenta que el número total de marcos de bloque es de doce, se deduce que la presión de memoria para la que está configurado es aproximadamente del 83%. Para aumentar o disminuir la presión basta con modificar el número de etiquetas diferentes en todos los nodos.

Información de reemplazo : almacena el estado y la etiqueta de cada uno de los marcos de bloque del resto de nodos del sistema. Este lugar aparece y se utiliza en casi todas las páginas del modelo.

Información de directorio : almacena el propietario cada uno de los bloques del sistema. Este lugar aparece y se utiliza en casi todas las páginas del modelo.

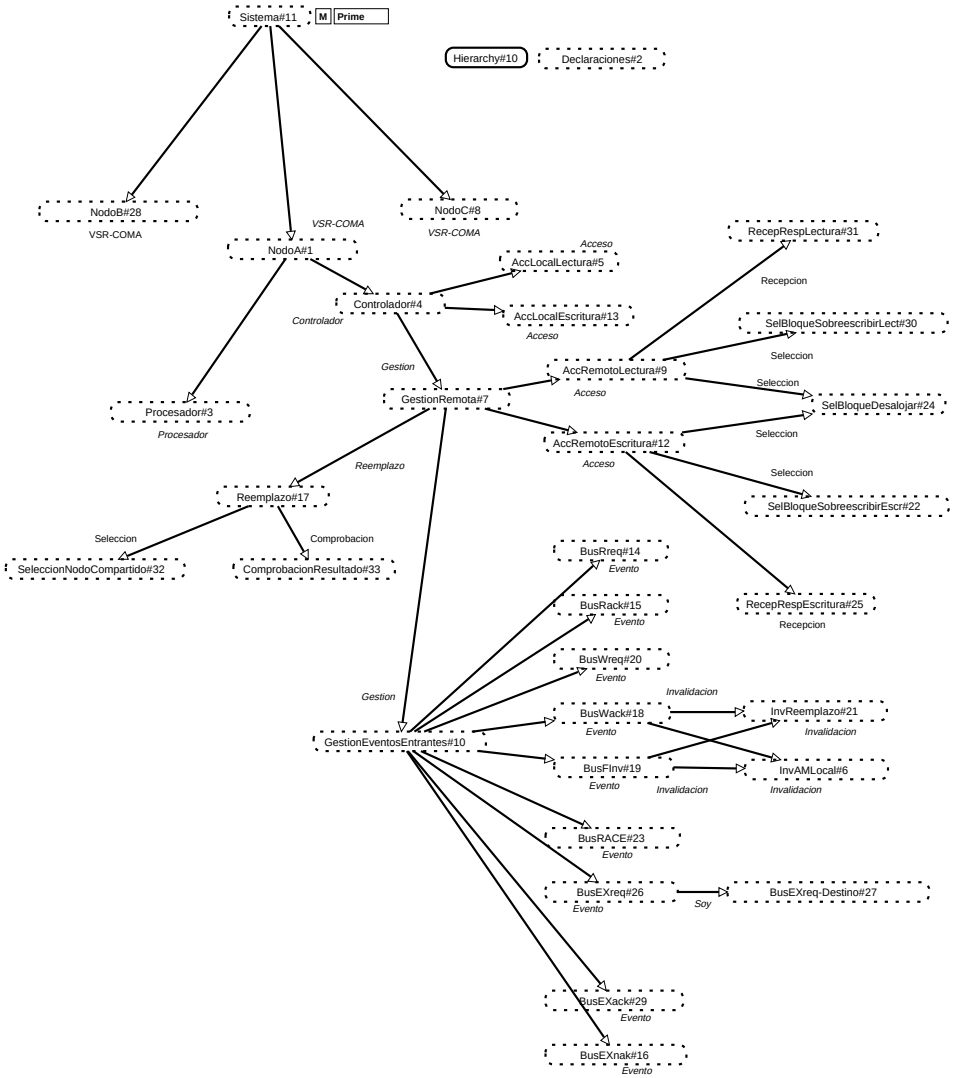


Figura A.4: Jerarquía de páginas de VSR-COMA.

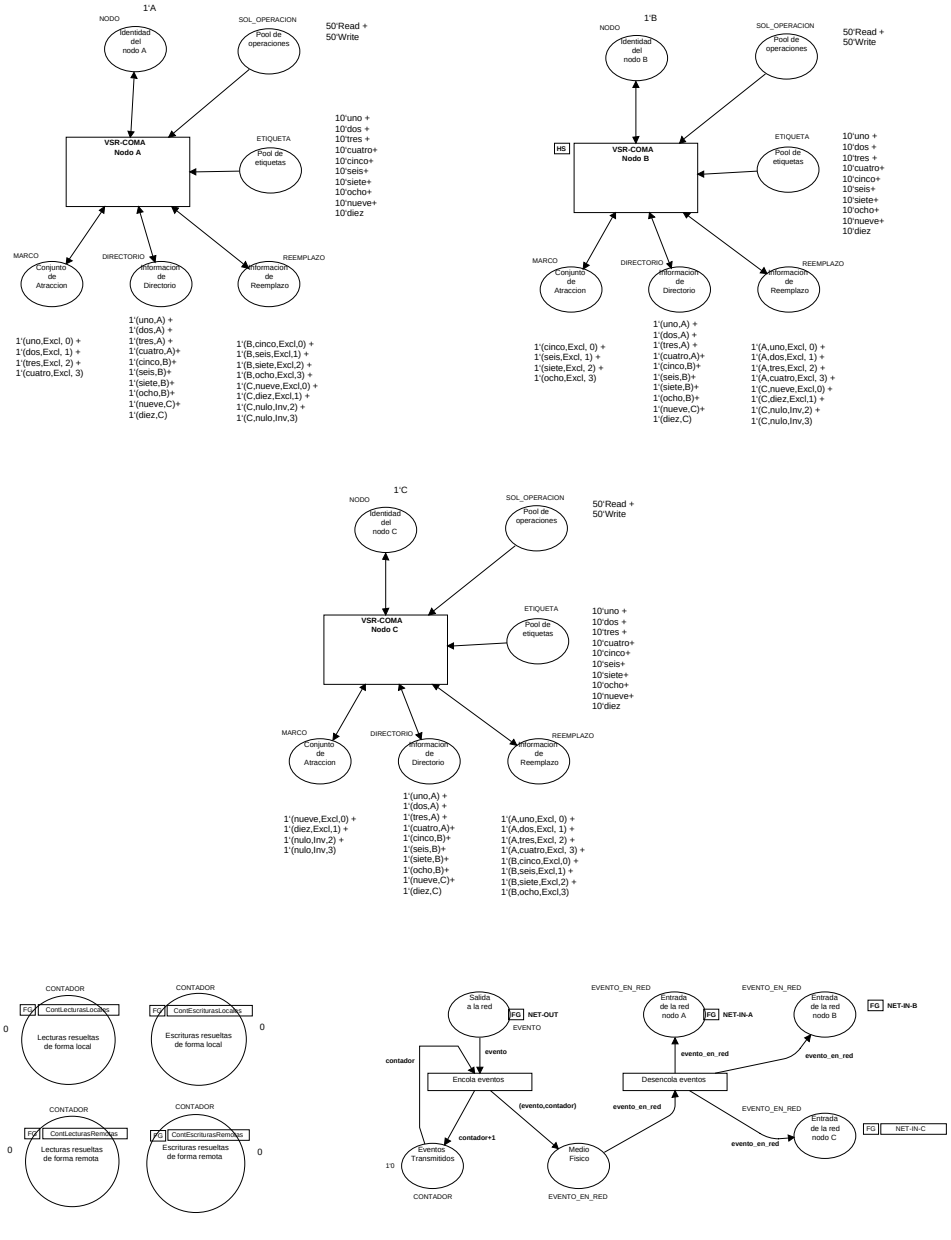


Figura A.5: Página “Sistema”.

Memoria de atracción : almacena el estado y la etiqueta de cada uno de los marcos de bloque de la memoria de atracción local. Este lugar aparece y se utiliza en casi todas las páginas del modelo.

Dado que el objetivo del modelo es la simulación del funcionamiento del protocolo, no la ejecución de una tarea concreta que requiera el uso de memoria compartida, no se ha modelado la existencia de los bloques concretos en cada una de las memorias de atracción: sólo se modela el estado y la etiqueta de cada uno de ellos, lo que resulta suficiente para los objetivos perseguidos.

Toda la información inicial presente en cada uno de los nodos de proceso de la figura A.5 debe ser coherente. Puede verse que la situación propuesta permite ejecutar una simulación compuesta de 300 operaciones sobre un total de diez etiquetas diferentes que cada nodo escogerá al azar, según el funcionamiento de las redes de Petri coloreadas.

Los cuatro contadores que aparecen en la parte inferior izquierda de la figura corresponden al número de lecturas y de escrituras resueltas de forma local y remota. Estos contadores son actualizados en las páginas correspondientes del modelo, y aparecen aquí para permitir al desarrollador observar simultáneamente el estado de la simulación a través de una única página. Al finalizar la simulación, la suma de los valores de estos cuatro lugares deberá ser igual al número total de operaciones realizadas en todos los nodos.

Finalmente, en la parte inferior derecha, puede verse la red que simula el funcionamiento del bus común. Existe una única salida a la red y tres entradas, correspondientes a cada uno de los nodos. Cuando un nodo decide enviar un evento, lo deposita en el lugar “Salida a la red”. A través de la transición “Encola eventos” el evento recibe un número único. Una vez transmitido, la transición “Desencola eventos” deposita una copia del evento en el buffer de entrada de cada uno de los nodos. El número de evento servirá para procesar en orden los eventos recibidos.

A.4 Jerarquía de páginas

A continuación detallaremos brevemente el cometido de cada una de las páginas básicas del modelo.

A.4.1 Página “Nodo A”

La página “Nodo A” (figura A.6) modela a gran escala el funcionamiento del nodo. Consta de dos transiciones de substitución, denominadas “Procesador” y “Controlador cache”. El funcionamiento es el siguiente. El procesador toma una operación y una etiqueta, y con ellas genera un dato de tipo OPERACION. Ese dato se transfiere al controlador cache, el cual, haciendo

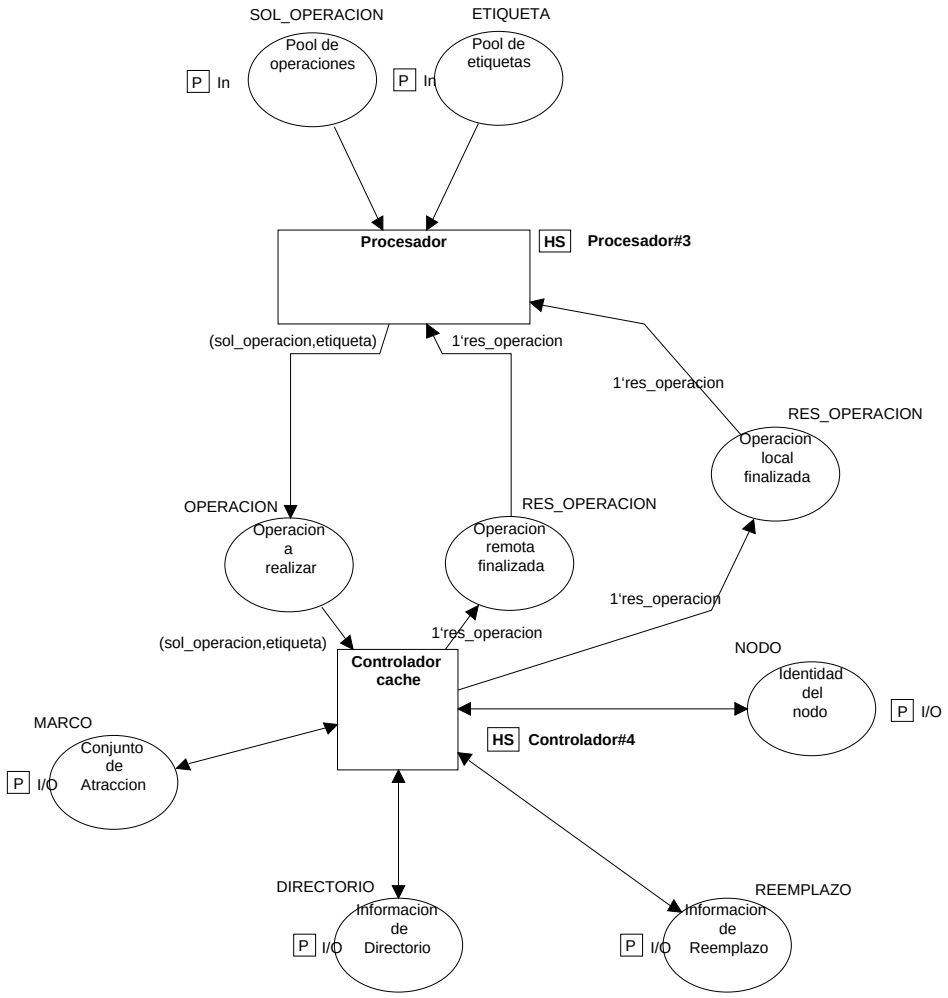


Figura A.6: Página “Nodo A”.

uso de la identidad del nodo, la información de reemplazo y directorio y la memoria de atracción local, resuelve la operación, devolviendo un dato que indica operación finalizada. Dependiendo de si la operación se ha podido resolver de una forma u otra, el dato se devolverá a través de “Operación local finalizada” u “Operación remota finalizada”. Cuando se reciba esta respuesta, el procesador generará una nueva solicitud. El proceso se repetirá hasta que los lugares que contienen las operaciones y las etiquetas se vacíen.

A.4.2 Página "Procesador"

La página "Procesador" (figura A.7) modela el comportamiento del procesador. Básicamente, la tarea del procesador será la siguiente:

1. Generar la siguiente operación.

2. Esperar a que no haya eventos pendientes de tratamiento.

3. Devolver a la página superior en la jerarquía la operación, a través del lugar "Operación a realizar" que posee un puerto de salida (parte inferior izquierda de la figura A.7).

4. Incrementar el contador de operaciones pendientes. Una vez hecho esto, se espera a que la operación generada finalice, bien sea de forma local o remota.

5. Si recibe un dato en el lugar "Operación remota finalizada", realiza las siguientes acciones:

   (a) Almacena la confirmación de la operación en el "Depósito de ACKs recibidos".

   (b) Incrementa el contador de operaciones realizadas.

   (c) Devuelve un testigo a través del lugar "Testigo de eventos entrantes". Este testigo permitirá continuar el tratamiento de los eventos pendientes, que tienen mayor prioridad que las operaciones.

   (d) Genera un testigo que permitirá generar una nueva operación.

6. Si recibe un dato en el lugar "Operación local finalizada", realiza las siguientes acciones:

   (a) Incrementa el contador de operaciones realizadas.

   (b) Genera un testigo para decrementar el contador debido a operaciones locales.

   (c) Genera un testigo que permitirá generar una nueva operación.

   En este caso no se devuelve un testigo en "Testigo de eventos entrantes", ya que no se ha necesitado acceder a la red y ese testigo no se ha consumido, por lo que no hace falta reponerlo.

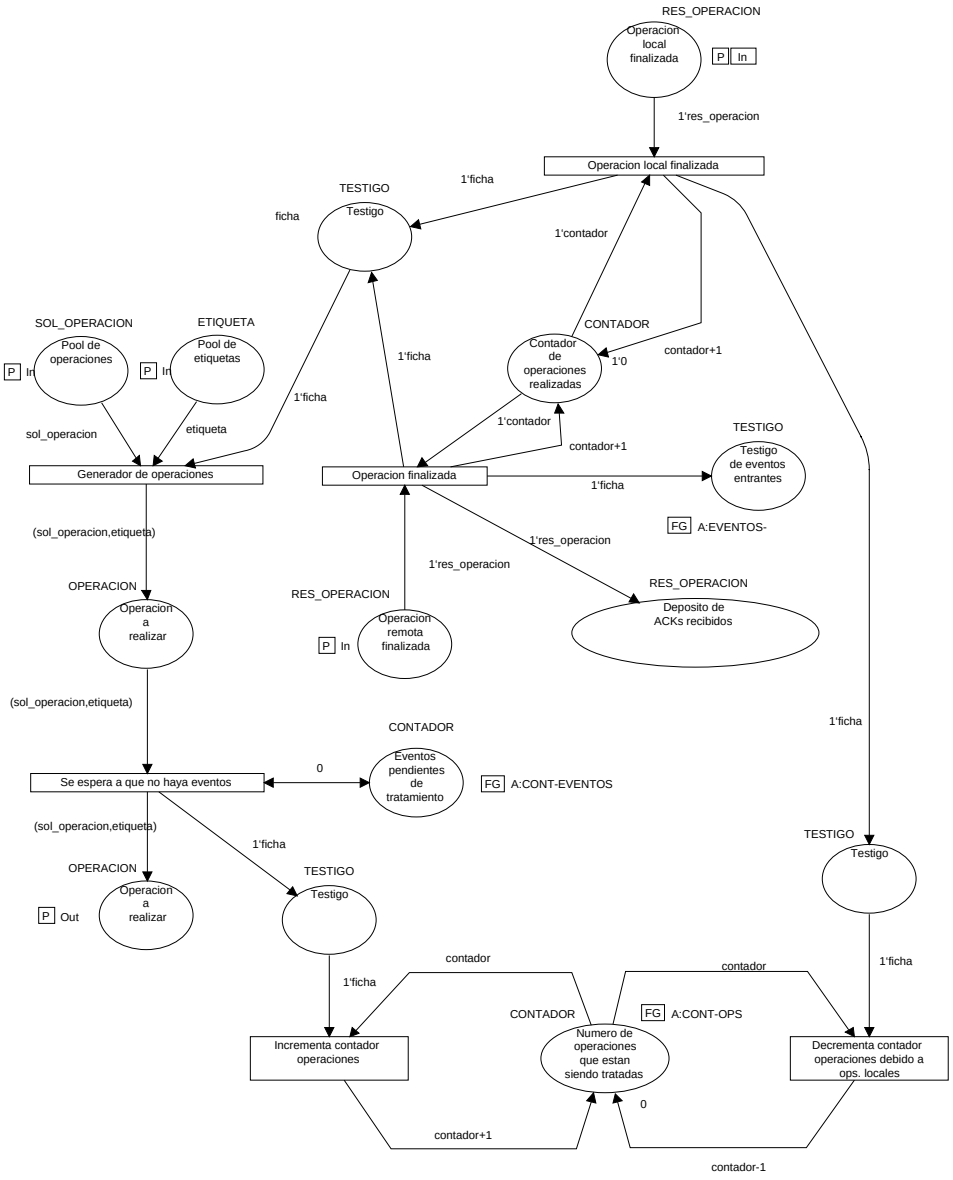


Figura A.7: Página “Procesador”.

A.4.3 Página "Controlador"

La página "Controlador" (figura A.8) modela el comportamiento del controlador cache. A grandes rasgos, el controlador realiza las siguientes acciones.

1. Si la operación solicitada es una lectura, el control se transfiere a la página "AccLocalLectura", que intentará resolverla de forma local. Si lo consigue, esta página devuelve un token de "operación local finalizada". En caso contrario, devuelve un token en "testigo de acceso remoto".

2. Si la operación solicitada es una escritura, el control se transfiere a la página "AccLocalEscritura", que intentará resolverla de forma local. Si lo consigue, esta página devuelve un token de "operación local finalizada". En caso contrario, devuelve un token en "testigo de acceso remoto".

3. Si aparece un token en "testigo de acceso remoto", indicando que la operación no puede resolverse de forma local, se transfiere el control a la página "GestionRemota". Cuando la operación remota se realice, se devuelve un testigo a través del lugar de salida "Operación remota finalizada".

Puede darse el caso de que la operación pueda resolverse de forma local aunque sea necesario enviar un evento de invalidación rápida a la red, por ejemplo para escribir un dato en SharOwn. En este caso, la transición "Gestión remota" devolverá un testigo en "Operación local finalizada", pese a haber generado un evento.

A.4.4 Página "Acceso local en lectura"

La página "Acceso Local en Lectura" (figura A.9) modela el comportamiento del controlador cache para el caso de una solicitud de lectura que pueda resolverse de forma local. La operación a realizar entra a esta página a través del lugar del mismo nombre (en la parte superior izquierda de la figura). Si el dato está presente en la memoria de atracción, se dispara la transición "Acceso local en lectura", que incrementa el contador de lecturas locales y genera la respuesta para el procesador. Además, esta respuesta dispara la transición "Solicitud finalizada", que permite eliminar el testigo de solicitud pendiente que bajará por los puntos de paso 1 y 2.

En el caso de que la lectura no pueda hacerse de forma local, ya que las condiciones no se cumplen en la memoria de atracción, la transición "Acceso local en lectura" no se disparará. En ese caso, el testigo que aparece en el punto de paso 2 no se elimina a través del disparo de "Solicitud finalizada", ya que no habrá ningún dato en ese lugar. Por el contrario, se

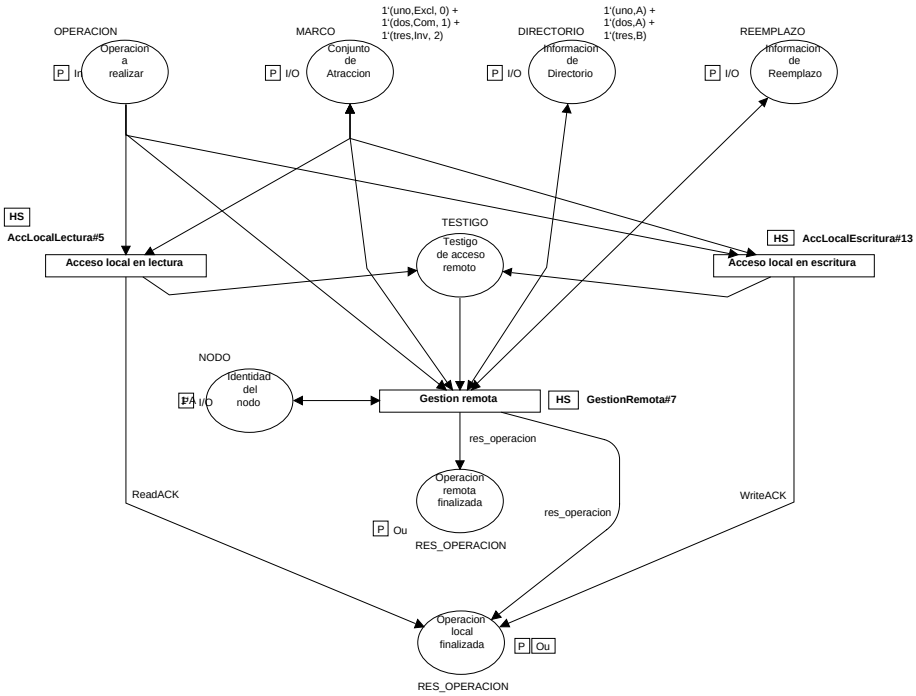


Figura A.8: Página “Controlador”.

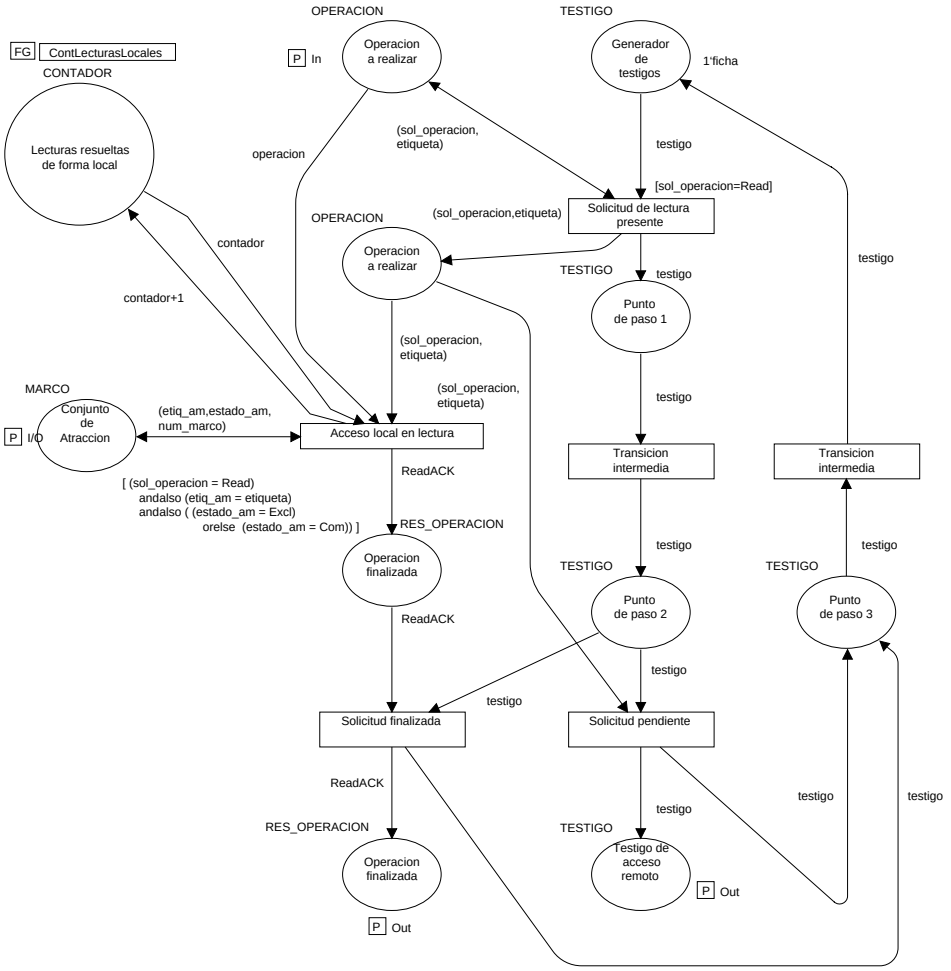


Figura A.9: Página “Acceso local en lectura”.

disparará “Solicitud pendiente”, ya que al no dispararse “Acceso local en lectura” el lugar “Operación a realizar” situado a su entrada seguirá conteniendo un dato. En este caso se generará un “Testigo de acceso remoto” que permitirá al controlador de coherencia activar la página de gestión remota.

A.4.5 Página “Acceso local en escritura”

La página “Acceso local en escritura” (figura A.10) modela el comportamiento del controlador cache para el caso de una solicitud de lectura que pueda resolverse de forma local. El funcionamiento es muy similar al descrito en la sección A.4.4. Sólo hay que tener en cuenta un detalle: la operación sólo se resolverá aquí si el dato está presente en la memoria de atracción local como “Exclusivo”. En caso contrario se activará la página de

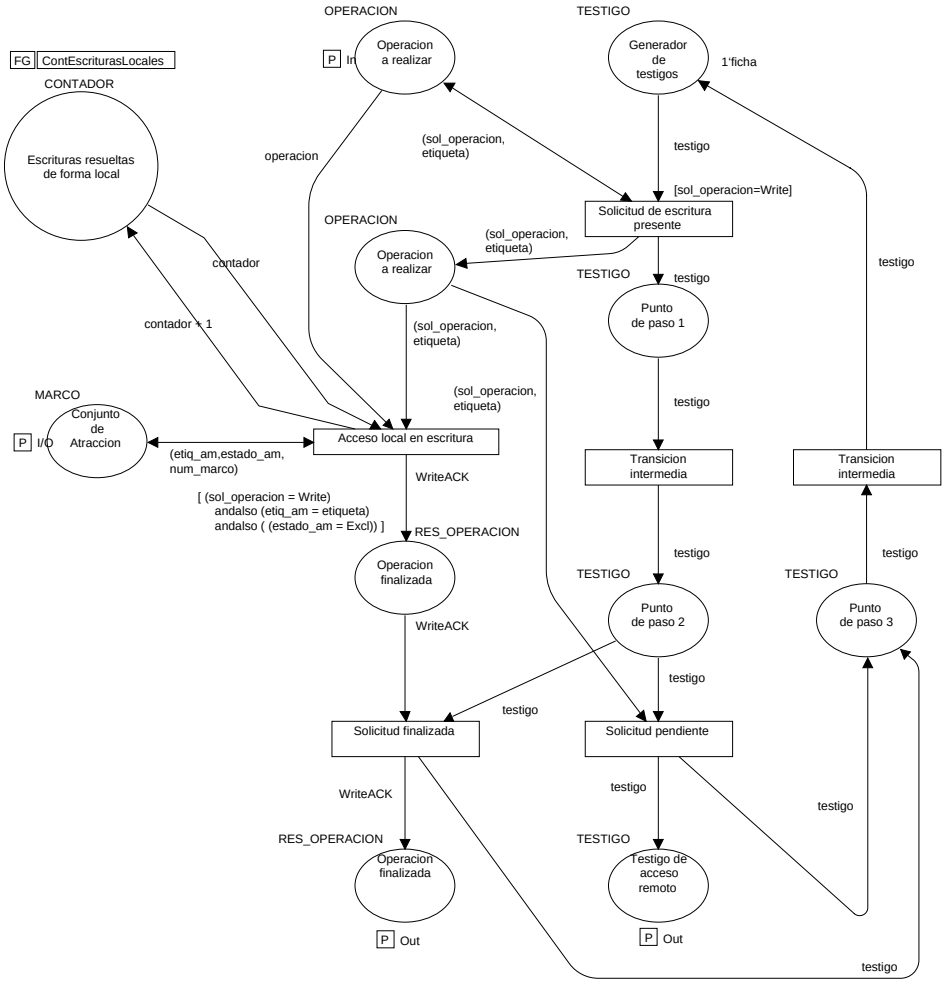


Figura A.10: Página “Acceso local en escritura”.

gestión remota, bien sea para obtener una copia exclusiva del dato o para obtener acceso exclusivo a una copia compartida de la que el nodo local es propietario.

A.5 Accesos remotos

A continuación veremos cómo se resuelven los accesos a bloques remotos, tanto para lectura como para escritura. En la sección A.7 veremos en detalle el tratamiento que se dispensa a los eventos que llegan a través de la red.

La figura A.11 muestra la estructura de la gestión que el controlador de coherencia realiza cuando debe obtener un bloque de forma remota para satisfacer una solicitud del procesador. Según la operación a realizar, se

activa la página de “Acceso remoto en lectura” o “Acceso remoto en escritura”. Estas dos páginas pueden devolver tres testigos diferentes: operación local finalizada, operación remota finalizada o número de marco a reemplazar. Este último dato indica que la operación no puede completarse porque no hay espacio en el conjunto de la memoria de atracción local, y que por lo tanto se hace necesario desalojar un bloque. El número del marco a reemplazar se introduce así en la página “Reemplazo”, que se encarga de gestionar el reemplazo.

Cuando el reemplazo finaliza, se devuelve un testigo de “Reemplazo finalizado”. Este testigo se utilizará para reiniciar la operación de lectura o escritura que ha quedado pendiente. Puede suceder que, tras el reemplazo, la lectura o escritura que antes era remota ahora sea local, ya que el bloque que se pretendía leer pudo haber sido desalojado desde otro nodo hacia el nodo local. En este caso, las páginas de lectura y escritura remota devolverán un testigo de “Operación local finalizada”. En caso contrario, al finalizar el reemplazo existirá un hueco en donde almacenar el bloque, por lo que el bloque podrá solicitarse, completando la operación pendiente.

A.5.1 Página “Acceso remoto en escritura”

La página “Acceso remoto en escritura” (figura A.12) modela el comportamiento del controlador de coherencia cuando se trata de solicitar de forma remota un bloque para su escritura. Los pasos que se realizan son los siguientes:

1. Se comprueba si el nodo local es propietario de ese bloque. Esto sólo puede suceder si el bloque está en estado “Compartido” y además el nodo local es propietario, lo que equivale al estado SharOwn. El nodo no puede tener el bloque en estado “Exclusivo” porque en ese caso no estaríamos hablando de una lectura remota sino local. Si el nodo es propietario del bloque se lanza un evento de invalidación rápida.

2. Si el nodo local no es propietario del bloque, selecciona un bloque de su memoria de atracción para almacenar el bloque que se va a solicitar, a través de la página “Selección del bloque a sobreescribir”.

3. Una vez seleccionado, se solicita el bloque a través de un evento de solicitud de escritura y se pasa a la página “Recepción de la respuesta de escritura” hasta que se reciba la respuesta. Esta página se encargará de devolver el testigo de “operación remota finalizada”.

4. Si no existe espacio para almacenar el bloque que va a solicitarse, se selecciona un bloque a desalojar y se lanza el número de marco a la página “Reemplazo”, a través del lugar de salida “Información de reemplazo”, a la derecha de la imagen.

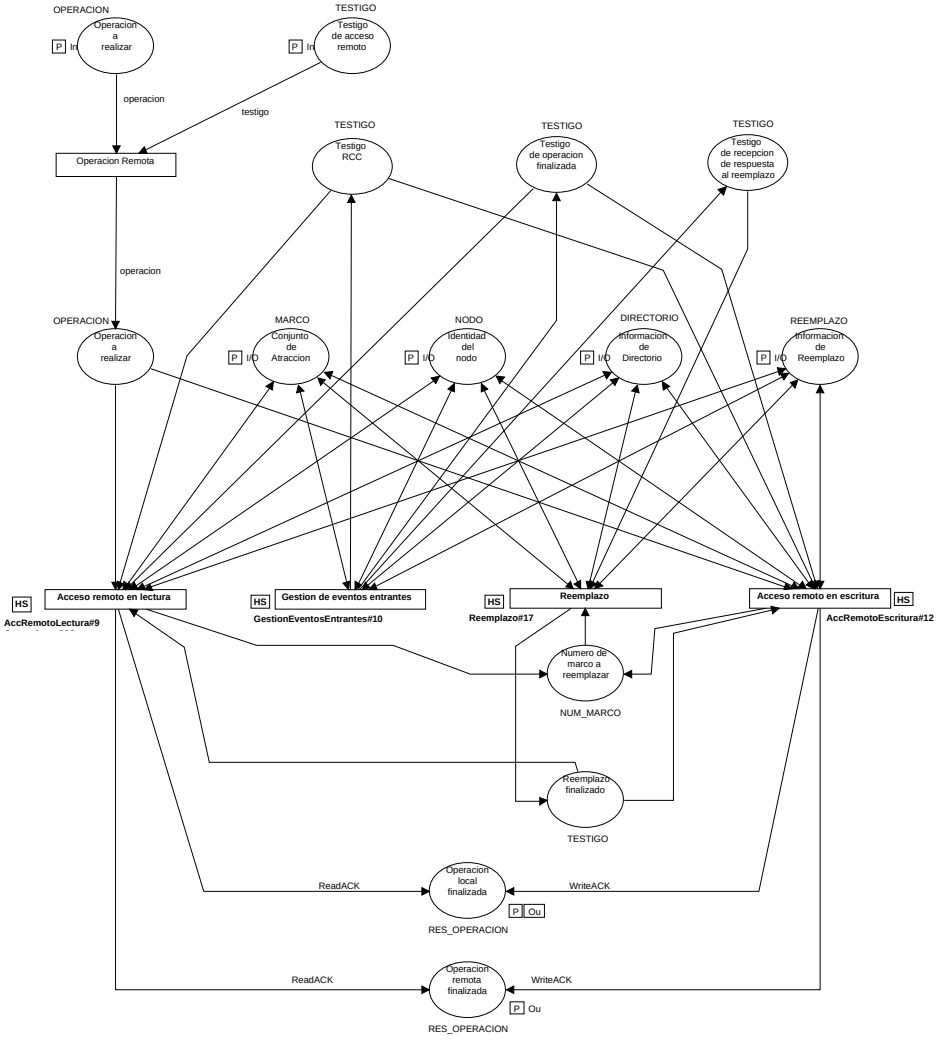


Figura A.11: Página “Gestión Remota”.

5. Cuando el reemplazo haya finalizado, se comprueba nuevamente si se dispone de una copia exclusiva del dato, producto de un eventual desalojo desde otro nodo mientras el nodo local intentaba generar un marco vacío para recibirlo.

6. Si no se dispone de una copia exclusiva, el procedimiento se reinicia comprobando la propiedad del bloque. La diferencia de este segundo intento con respecto al primero es que ahora sí existe un hueco en donde almacenar el bloque que va a solicitarse.

A.5.2 Página "Selección del bloque a sobreescribir en escritura"

La página "Selección del bloque a sobreescribir en escritura", que aparece en la figura A.13, modela el comportamiento del controlador de coherencia cuando se trata de seleccionar el marco de bloque que se utilizará para almacenar el bloque cuando se solicite.

El algoritmo de selección que se utiliza en el modelo de VSR-COMA implementado es básicamente el que se ha propuesto en la sección 4.7.2 del presente trabajo.

1. Si hay un bloque "Inválido" en la memoria de atracción, su estado cambia a "InvEsperandoExcl" (estado equivalente al estado InvWExcl descrito en el protocolo) y se envía la solicitud de escritura a la red.

2. Si no hay bloques inválidos, se busca alguno en estado "Compartido" del cual el nodo local no sea propietario y se lo sobreescribe, procediéndose igual que en el punto anterior.

   En ambos casos se incrementa el contador que indica el número de operaciones que están siendo tratadas.

3. En caso contrario, se hace necesario desalojar un bloque, para lo cual se devuelve la etiqueta a leer a la página "Acceso remoto en escritura", que se encargará de activar el mecanismo de selección del nodo a desalojar para iniciar así el reemplazo.

A.5.3 Página "Selección del bloque a desalojar"

Como puede verse en la jerarquía de páginas (figura A.4), la página "Selección del bloque a desalojar" es compartida por las páginas "Acceso remoto en escritura" y "Acceso remoto en lectura". La razón es que la selección del bloque a desalojar es independiente del uso al que se destine el marco que el bloque ocupe.

La página puede verse en la figura A.14. El algoritmo de selección es el desarrollado en la sección 4.7.3 del presente trabajo.

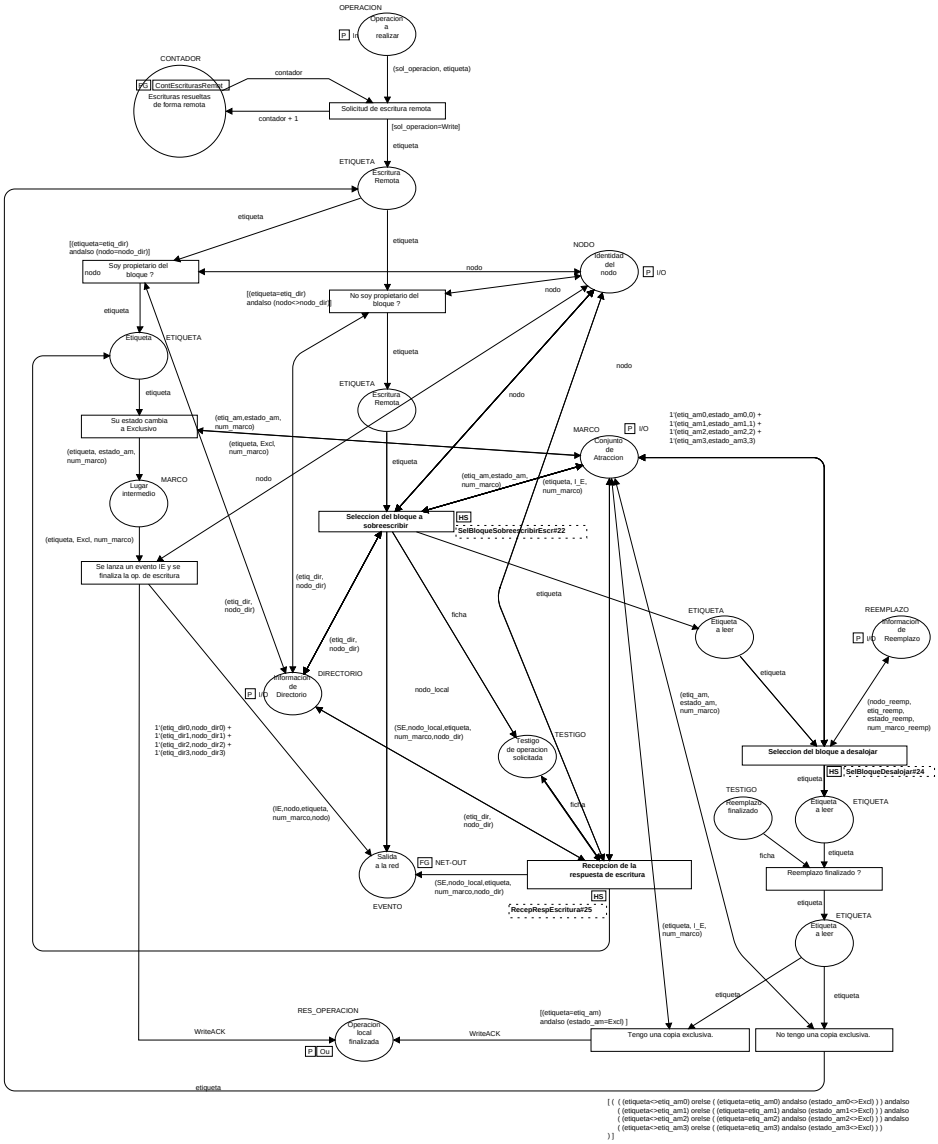


Figura A.12: Página “Acceso remoto en escritura”.

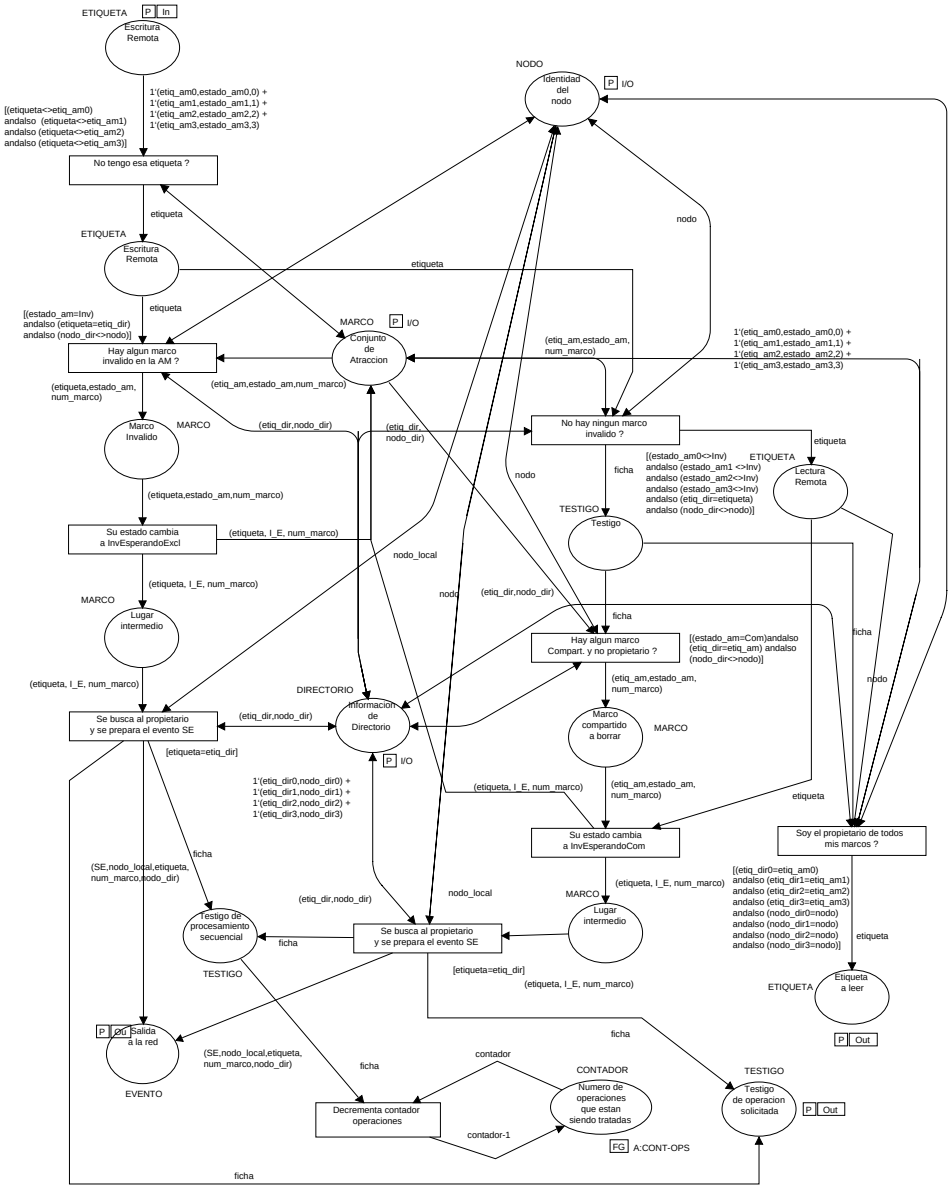


Figura A.13: Página “Selección de bloque a sobreescribir en escritura”.

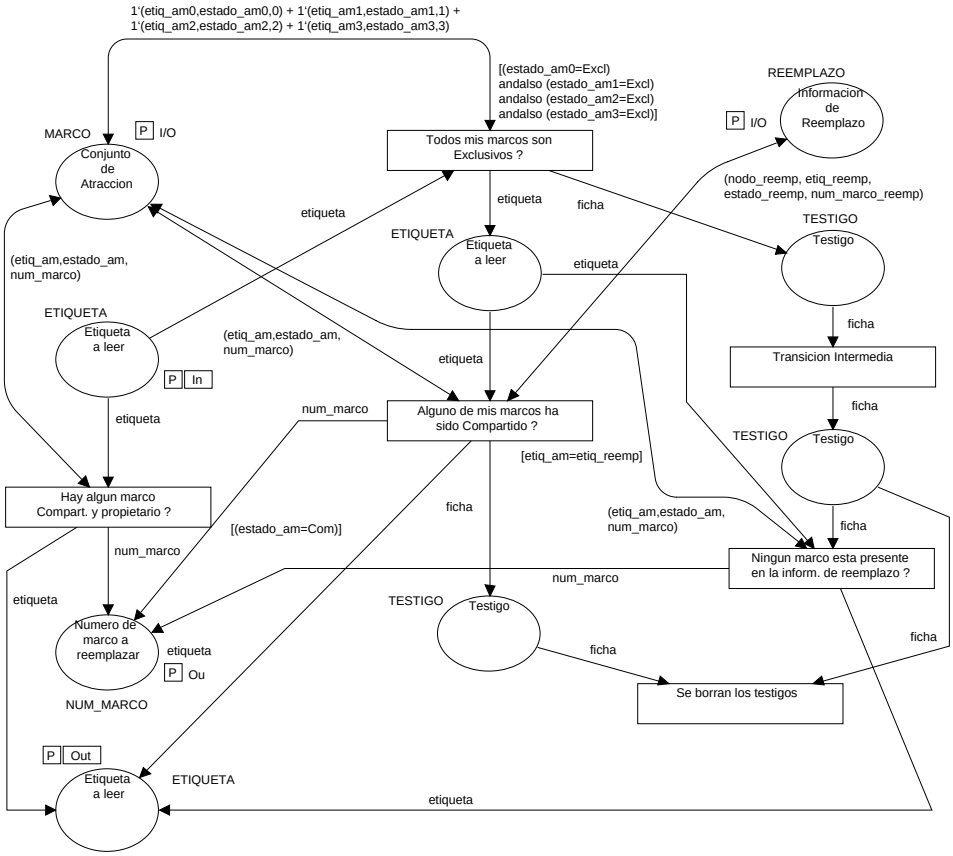


Figura A.14: Página “Selección de bloque a desalojar”.

1. Se comprueba si algún marco está en estado “Compartido” y el nodo local es el propietario. De existir alguno, se lo selecciona para el desalojo.

2. Si no hay ninguno en estas condiciones, se selecciona un bloque en estado “Exclusivo” y comprueba si alguno de los marcos del nodo local ha sido compartido por otros nodos.

3. En caso contrario, se selecciona un bloque al azar.

A.5.4 Página “Recepción respuesta de escritura”

La página “Recepción respuesta de escritura”, que aparece en la figura A.15, modela el comportamiento del controlador de coherencia cuando se trata de esperar la respuesta a una solicitud de escritura remota. Cuando se activa el “testigo de operación solicitada” el sistema espera la respuesta a

través de la gestión de los eventos entrantes. Pueden darse tres situaciones: que aparezca el "testigo de operación finalizada", lo que dará fin a la operación de escritura y permite devolver el dato al procesador; que aparezca el "testigo RCC" (Resolución de Condiciones de Carrera), que indica que se ha recibido una respuesta que señala la existencia de una condición de carrera al hacer la solicitud, lo que lleva a reiniciarla nuevamente.

La tercera situación que puede darse consiste en la aparición del testigo "Etiqueta pendiente de carrera SL/SE vs SD". Este testigo indica que, mientras se esperaba la respuesta a la solicitud de escritura, el bloque solicitado ha llegado como producto de una solicitud de reemplazo por parte del nodo propietario. Esto hace que ya se disponga de una copia del bloque, aunque no se encuentre en estado "Exclusivo" sino "Compartido Propietario". En este caso, la página "Recepción respuesta de escritura" devuelve la etiqueta a su página superior en la jerarquía, "Acceso remoto en escritura", para que se reinicie la operación de escritura y se genere un evento de invalidación rápida, al objeto de obtener la exclusividad en el acceso. Esto vuelve a incrementar el contador de "operaciones que están siendo tratadas", que fue decrementado por la gestión de eventos entrantes, para indicar que no pueden seguir procesándose los eventos de entrada hasta que no se finalice con la operación en curso.

A.5.5 Página "Acceso remoto en lectura"

La página "Acceso remoto en lectura", que aparece en la figura A.16, modela el comportamiento del controlador de coherencia cuando se trata de obtener de forma remota un bloque para su lectura. Su funcionamiento es muy similar al descrito para el acceso remoto en escritura (sección A.5.1), aunque con algunas simplificaciones. El problema en este caso consiste en obtener una copia compartida del bloque, por lo que no es necesario garantizar un acceso exclusivo. Por lo tanto, si esa etiqueta no aparece en la memoria de atracción en estado "Exclusivo" o "Compartido" se selecciona un bloque a sobreescribir y se espera la respuesta de lectura. En caso de que no pueda sobreescribirse ningún bloque de la memoria de atracción, se inicia el desalojo de un bloque. Cuando se complete se comprueba que no haya aparecido una copia válida en la memoria de atracción, y se solicita el bloque en lectura a su propietario.

A.5.6 Página "Recepción respuesta de lectura"

La página "Recepción respuesta de lectura", que aparece en la figura A.17, modela el comportamiento del controlador de coherencia cuando se trata de esperar la respuesta a una solicitud de lectura remota. Su funcionamiento es similar al de la página "Recepción de respuesta de escritura", descrito en la sección A.5.4.

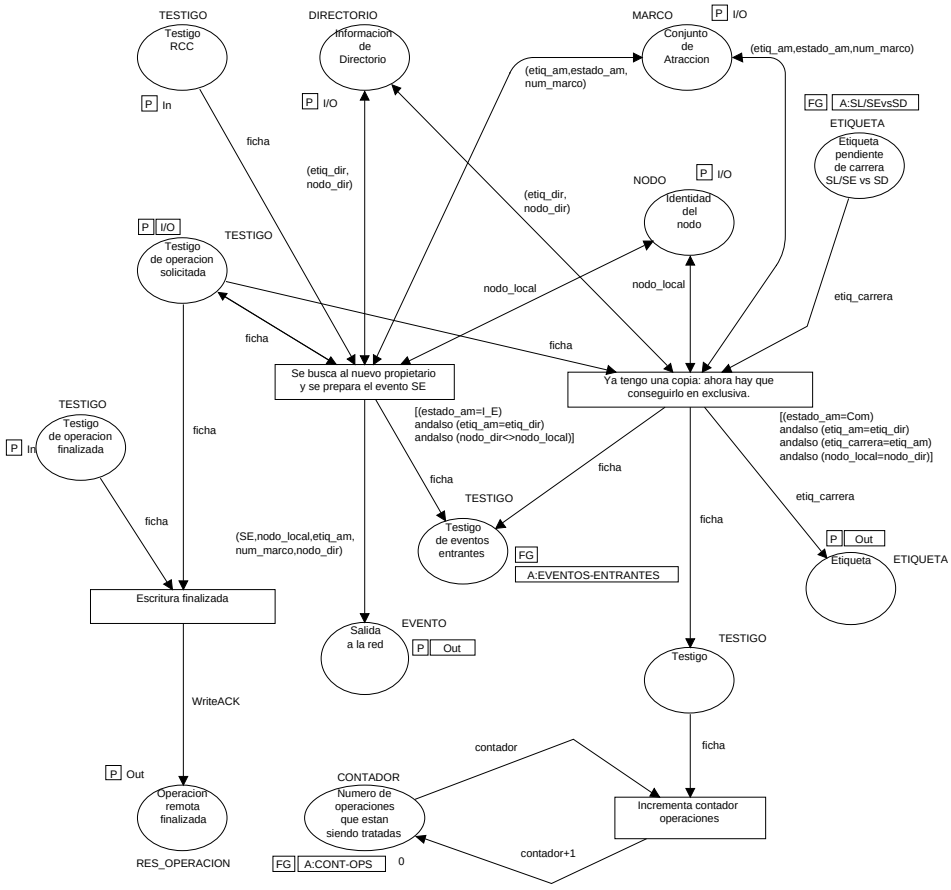


Figura A.15: Página “Recepción de respuesta de escritura”.

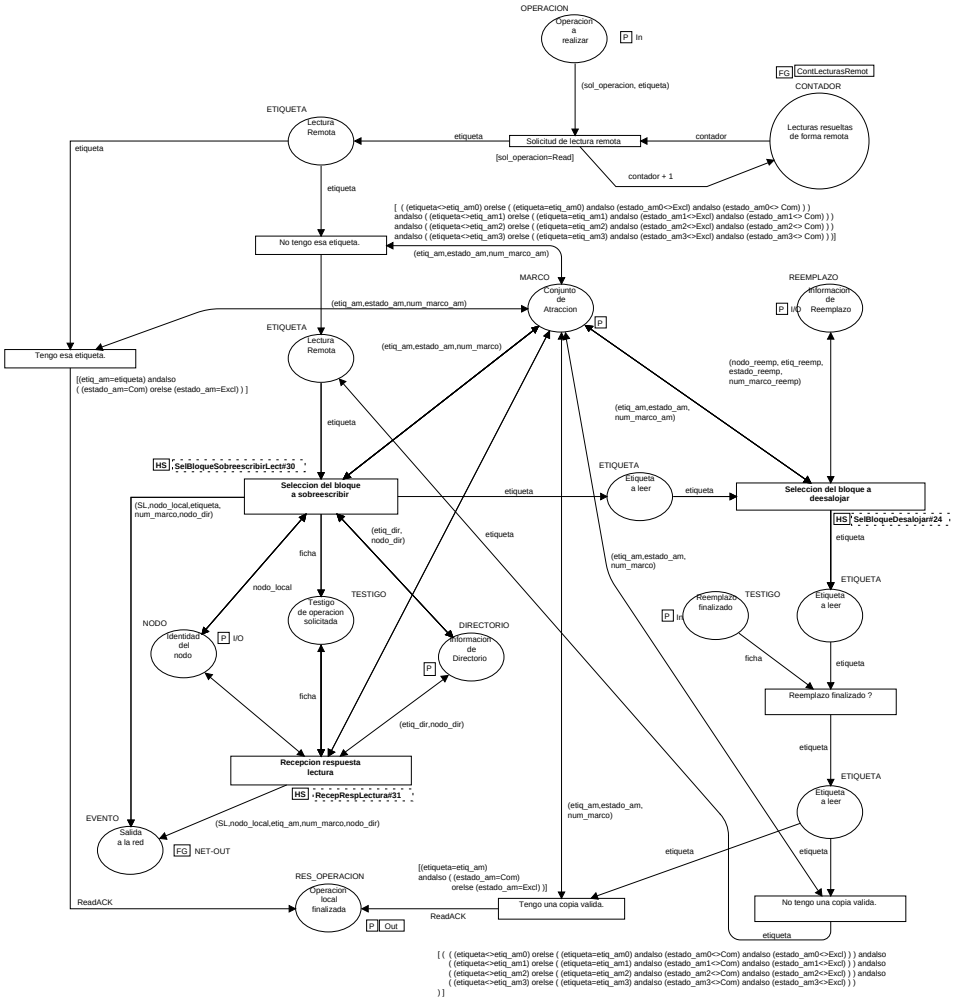


Figura A.16: Página “Acceso remoto en lectura”.

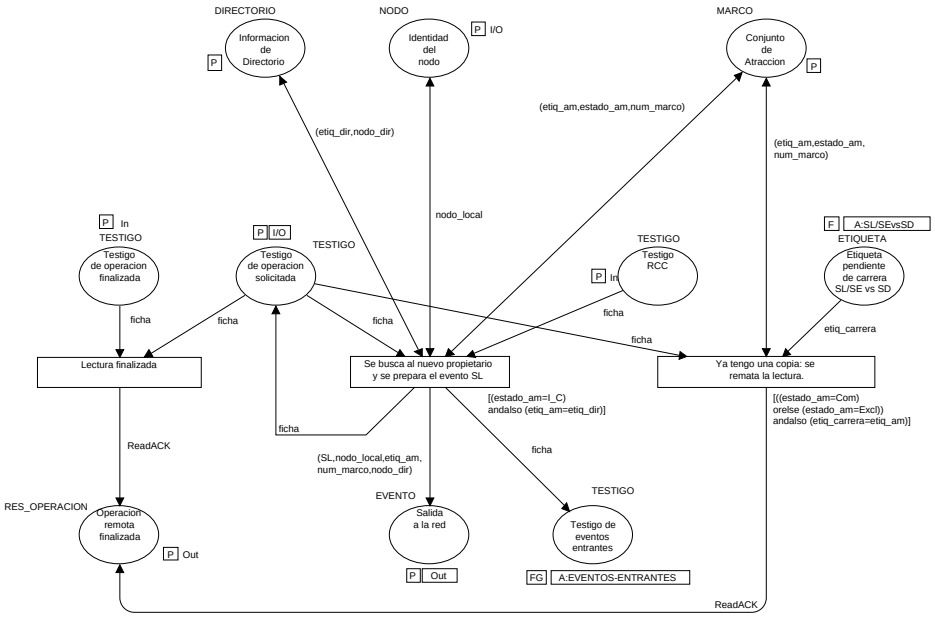


Figura A.17: Página “Recepción de respuesta de lectura”.

A.5.7 Página "Selección del bloque a sobreescribir en lectura"

La página "Selección del bloque a sobreescribir en lectura", que aparece en la figura A.18, modela el comportamiento del controlador de coherencia cuando se trata de seleccionar un marco de bloque que almacene un bloque para su lectura. Esta página se diferencia de la página "Selección del bloque a sobreescribir en escritura" (sección A.5.2) en que, una vez seleccionado el marco de bloque a sobreescribir, el evento generado será una solicitud de lectura en lugar de una de escritura.

A.6 Reemplazo

La página "Reemplazo" (figura A.19) muestra el comportamiento del modelo cuando deba realizarse una operación de desalojo sobre un marco de bloque concreto del cual el nodo local es propietario.

Como se ha expuesto en la sección 4.10, la estrategia de selección del nodo destino no está ligada al protocolo propiamente dicho. Por este motivo, la estrategia utilizada en el modelo con redes de Petri de VSR-COMA es una simplificación de la propuesta en la sección 4.7.3, dado que con este modelo no se pretende una selección óptima del nodo destino sino validar el funcionamiento del protocolo.

La estrategia de selección utilizada por el modelo es la siguiente:

1. Se comprueba si el estado del bloque es "Compartido", siendo el nodo local el propietario. Si es así, se comprueba si queda alguna copia en otro nodo. En ese caso, se envía el bloque a dicho nodo.

2. Si no quedan copias, o bien el bloque es "Exclusivo", se comprueba si existe algún marco "Inválido" en otros nodos. En ese caso, se envía el bloque a uno de ellos.

3. Si no existen marcos inválidos, se busca un marco "Compartido no Propietario".

Para desalojar el bloque se genera un evento de solicitud de desalojo (evento BusEXreq), que se envía al lugar de salida hacia la red. El marco correspondiente al bloque en la memoria de atracción pasa al estado "Esperando Desalojo". Además, se decrementa el contador de operaciones que están siendo tratadas, al objeto de permitir el tratamiento de los eventos entrantes y así poder recibir la respuesta a la solicitud de desalojo. La transición de substitución "Comprobación del resultado del reemplazo" se encarga de decidir la acción a realizar cuando se reciba la respuesta a la solicitud y, en su caso, modificar el estado de la copia que aún permanece en la memoria de atracción local.

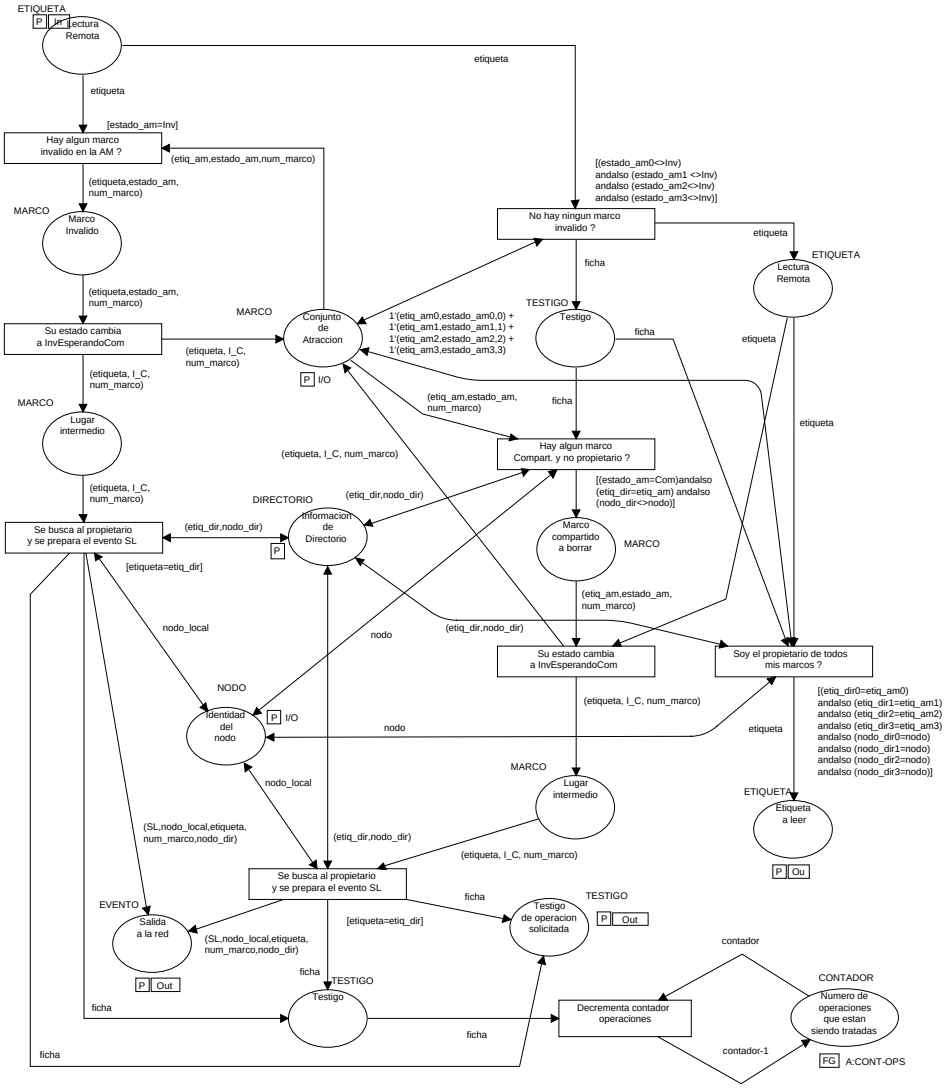


Figura A.18: Página “Selección de bloque a sobreescribir en lectura”.

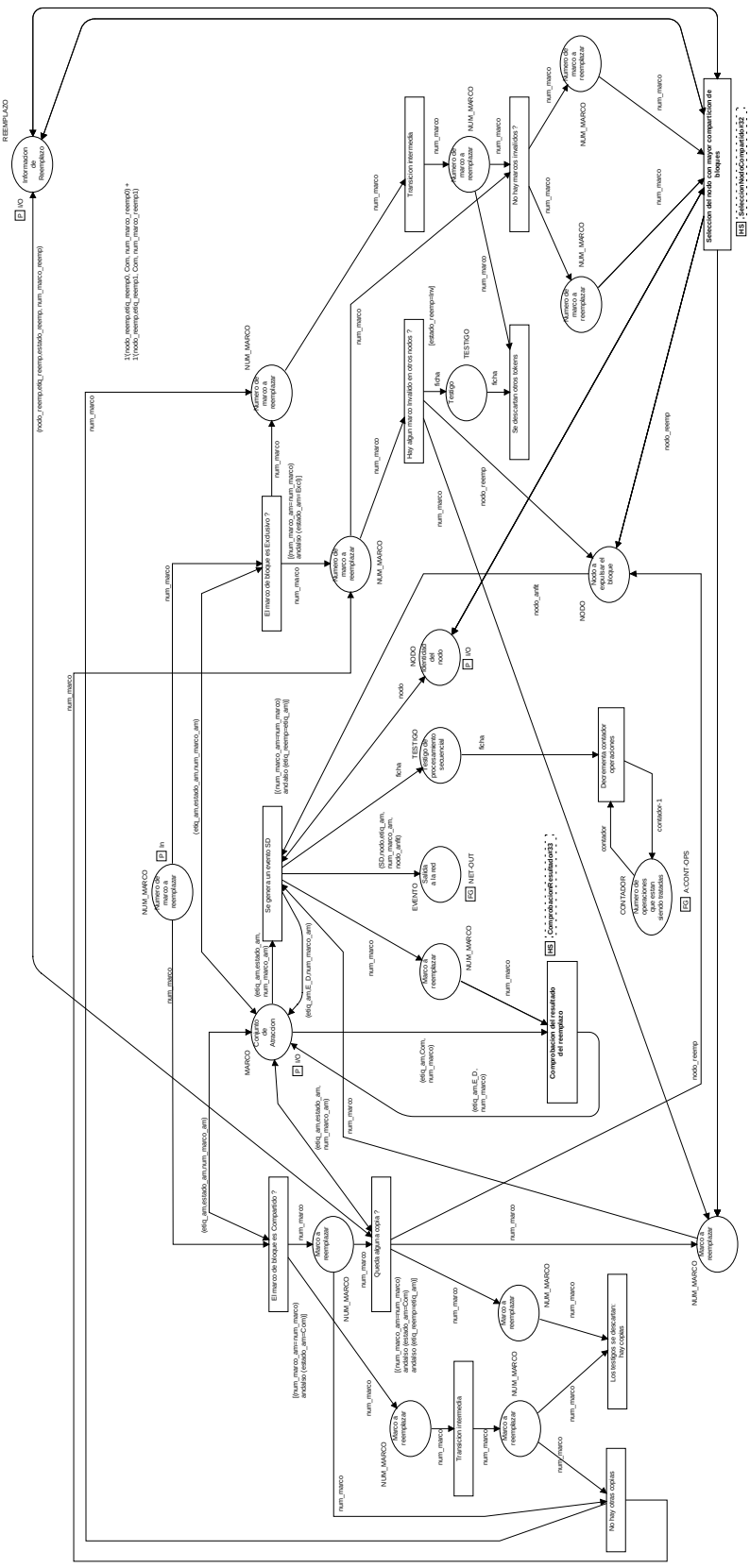


Figura A.19: Página “Reemplazo”.

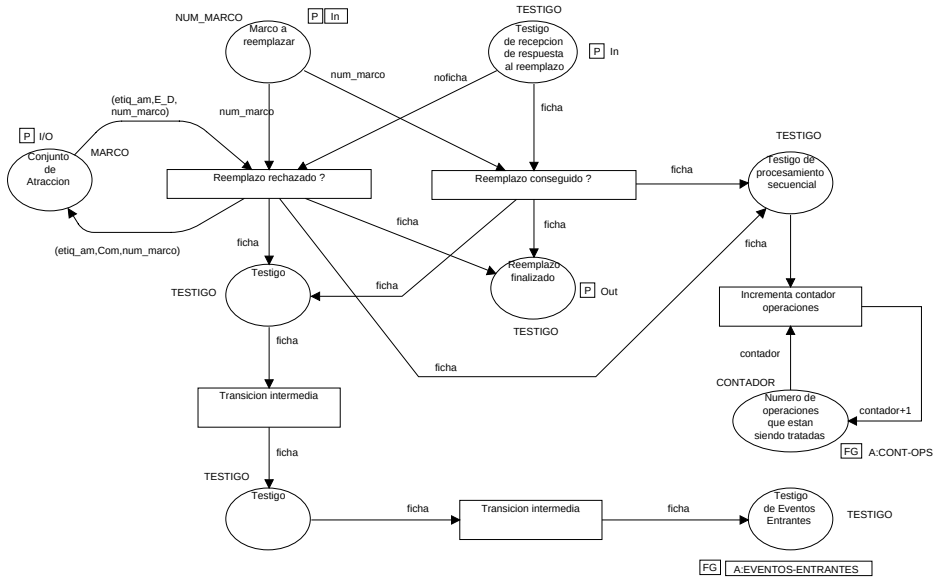


Figura A.20: Página “Comprobación Resultado”.

A.6.1 Página “Comprobación Resultado”

La página “Comprobación Resultado” (figura A.20) se encarga de modelar el comportamiento del controlador de coherencia al recibir la respuesta a una solicitud de reemplazo generada por el nodo local. Si el reemplazo ha sido rechazado, se cambia el estado del bloque que se ha intentado desalojar a “Compartido”, y se da por finalizada la operación de reemplazo (el nodo local sigue siendo propietario del bloque). El modelo se encargará entonces de comprobar si es necesario intentar otra operación de desalojo o bien si se ha producido alguna invalidación que haya generado un marco de bloque libre en el conjunto correspondiente de la memoria de atracción.

Si la operación de desalojo ha tenido éxito, la página “BusEXack” (que describiremos en la sección A.7.10) se habrá encargado de invalidar la copia local del bloque.

A.6.2 Página “Selección del nodo compartido”

La página “Selección del nodo compartido” (figura A.21) se encarga de modelar el comportamiento del controlador de coherencia a la hora de seleccionar un nodo con el mayor número posible de bloques en estado “Compartido” de los cuales el nodo no sea propietario. El objetivo de esto es balancear la carga de bloques en propiedad en cada nodo, intentando desalojar el bloque hacia un nodo que tenga pocos bloques en propiedad en ese conjunto. En el modelo, se comprueba si hay algún nodo con cuatro

bloques en estado “Compartido no Propietario”; en caso contrario, se busca uno con tres bloques; si falla, uno con dos; finalmente, se selecciona un nodo con un bloque en ese estado. Si la presión de memoria es menor que el 100 %, se garantiza la existencia de al menos un bloque que posea un marco de bloque ocupado por un bloque del cual el nodo no es el propietario.

Esta página devuelve el identificador del nodo hacia el cual se intentará desalojar el bloque.

A.7 Gestión de eventos entrantes

La gestión de los eventos entrantes es una parte muy importante del protocolo VSR-COMA. El modelo desarrollado con redes de Petri sigue el esquema utilizado en la descripción de la gestión hecha en la sección 4.9.

La página “Gestión de eventos entrantes”, que aparece en la figura A.22, modela el comportamiento del controlador de coherencia al recibir un evento a través del buffer de entrada procedente de la red (ver figura A.5). Una vez recibido el evento, se incrementa el contador de eventos pendientes de tratamiento, para indicar que existen eventos en el buffer y que no pueden iniciarse nuevas operaciones de memoria hasta no haberlos procesado todos (momento en el cual dicho contador valdrá cero). El evento se desencola basándose en su número de evento, al objeto de procesarlos en el orden en que se han recibido. Según el tipo de evento a tratar, se accede a la página de tratamiento correspondiente.

Existen diferentes testigos que se activarán en función del tratamiento. El testigo que aparece en “Salida del testigo de operación finalizada” indica que se ha recibido la respuesta a una solicitud de lectura o de escritura. Este testigo se utilizará para completar la operación en la página correspondiente (que será “Recepción de respuesta de lectura” o “Recepción de respuesta de escritura”, según el caso). El testigo almacenado en “Salida del testigo RCC” indica que se ha recibido una respuesta del tipo BusRACE. En este caso se reiniciará la operación correspondiente, utilizando la información de directorio actualizada.

El “Testigo de recepción de respuesta al reemplazo”, como su nombre indica, informa de que el reemplazo ha sido finalizado, independientemente de si ha tenido éxito o no. Este testigo se utilizará para volver a examinar la memoria de atracción local y decidir si es necesario iniciar nuevamente una operación de desalojo.

Cuando se ha acabado de procesar el evento entrante, aparecerá un testigo en el lugar “Salida del testigo secuencial”, indicando que puede continuarse el tratamiento de eventos. Como veremos, si el evento recibido es una respuesta a una solicitud hecha por el nodo local, este testigo desaparece hasta que la operación de memoria se completa. Cuando se haya completado, aparecerá un testigo en “Testigo de eventos entrantes” que

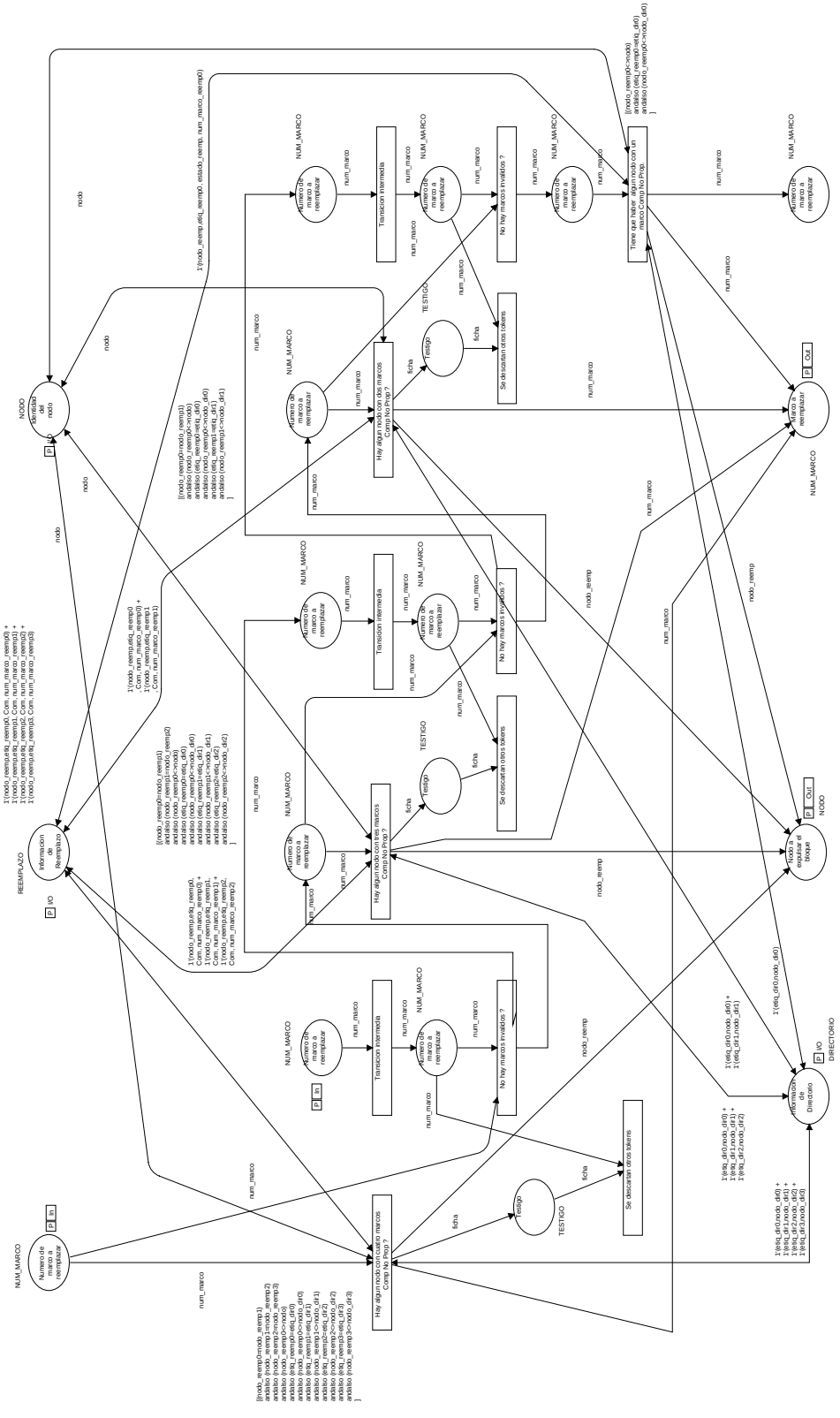


Figura A.21: Página “SeleccionNodoCompartido”.

permitirá seguir procesando eventos.

A.7.1 Página "BusRreq"

La página "BusRreq", que aparece en la figura A.23, modela el comportamiento del controlador de coherencia cuando se trata de procesar un evento entrante del tipo BusRreq.

Tras recibir este evento, los casos posibles son los descritos en la sección 4.9.2:

- El nodo local es origen del evento.
- El nodo local es destino y propietario.
- El nodo local es destino y no propietario.
- El nodo local es remoto.
- El nodo local es destino y propietario pero está pendiente una operación de desalojo.
- El nodo local es destino y propietario pero el bloque está en un estado transitorio.

A.7.2 Página "BusRack"

La página "BusRack", que aparece en la figura A.24, modela el comportamiento del controlador de coherencia cuando se trata de procesar un evento entrante del tipo BusRack.

Tras recibir este evento, los casos posibles son los descritos en la sección 4.9.3:

- El nodo local es origen del evento.
- El nodo local es destino del evento.
- El nodo local es remoto.

Sólo se genera el testigo secuencial en los casos 07 y 09: si el nodo local es destino del evento, se genera en su lugar el testigo de operación finalizada, al objeto de permitir la finalización de la operación de memoria antes de continuar con el tratamiento de los eventos entrantes.

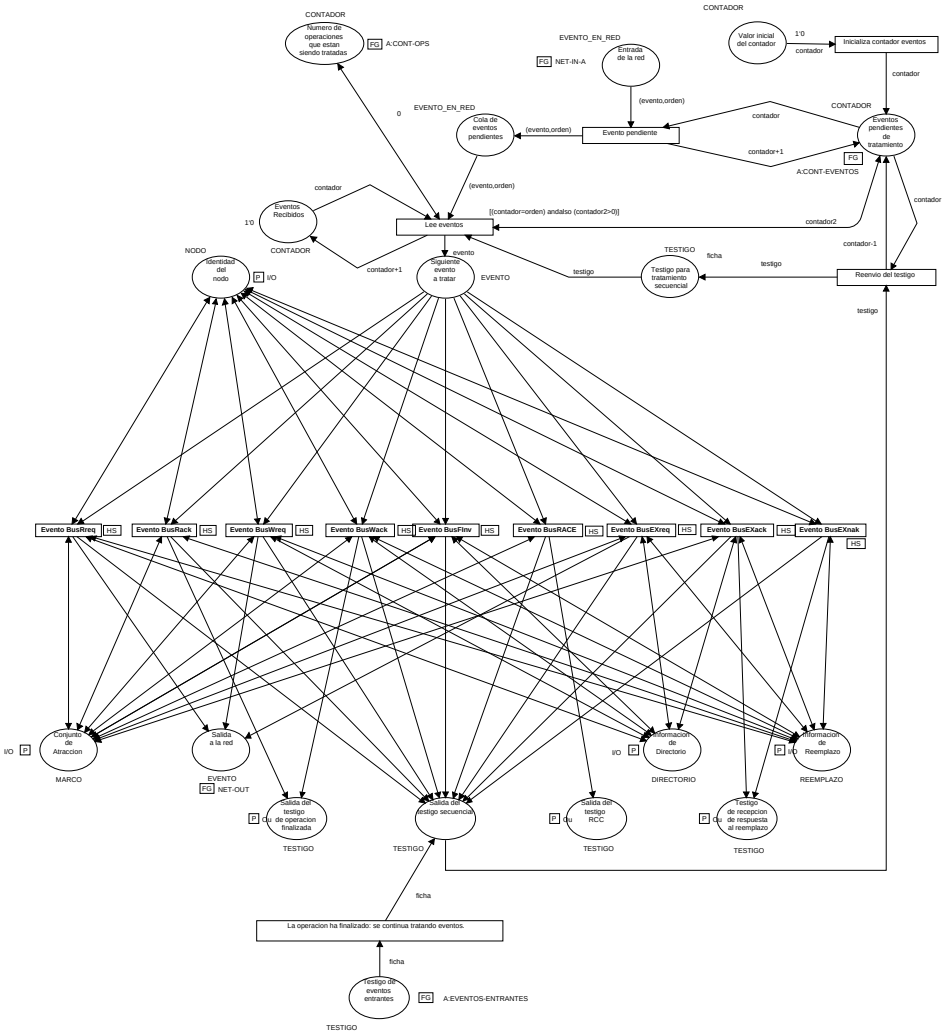


Figura A.22: Página “Gestión de eventos entrantes”.

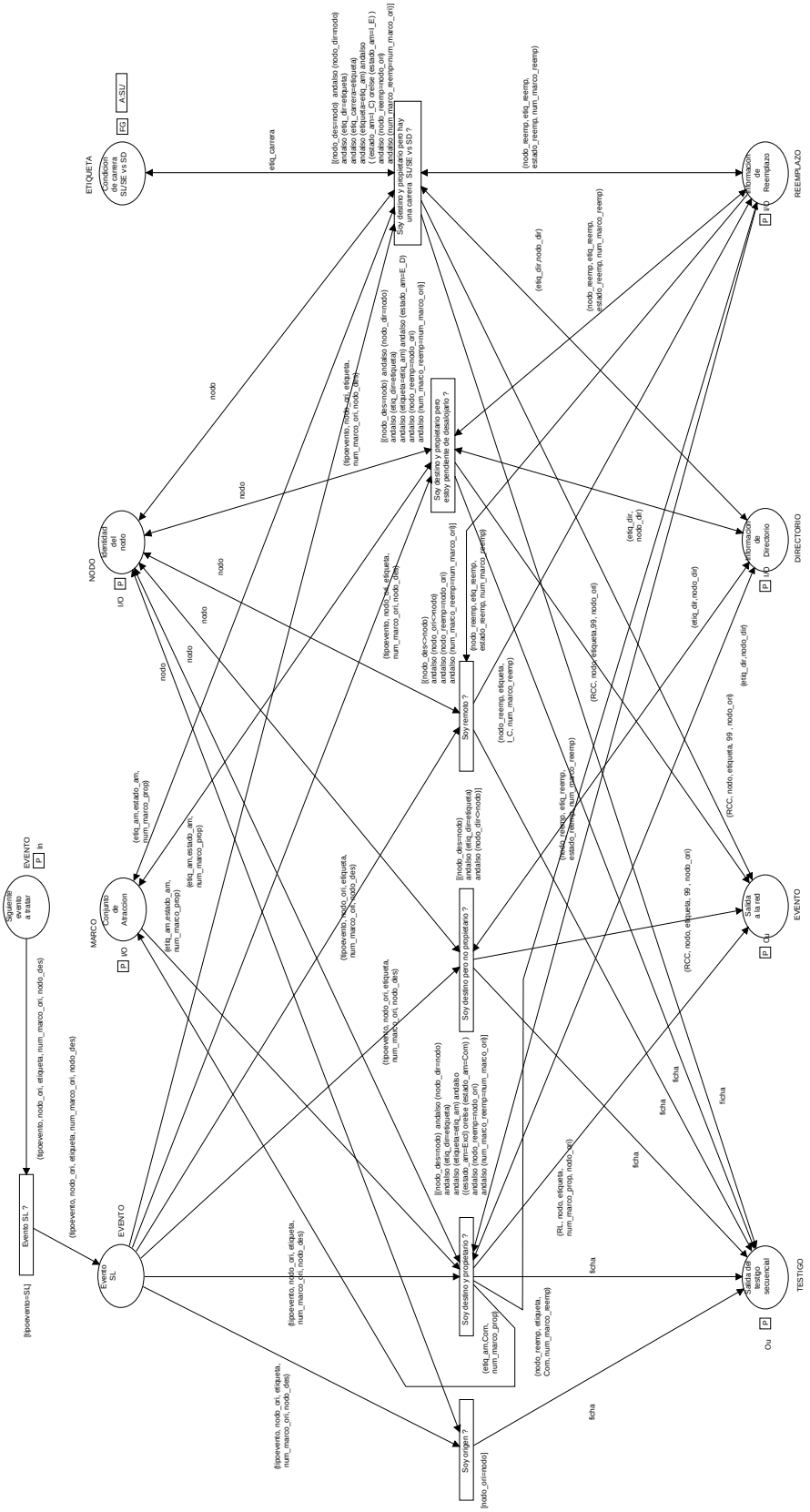


Figura A.23: Página “BusRreq”.

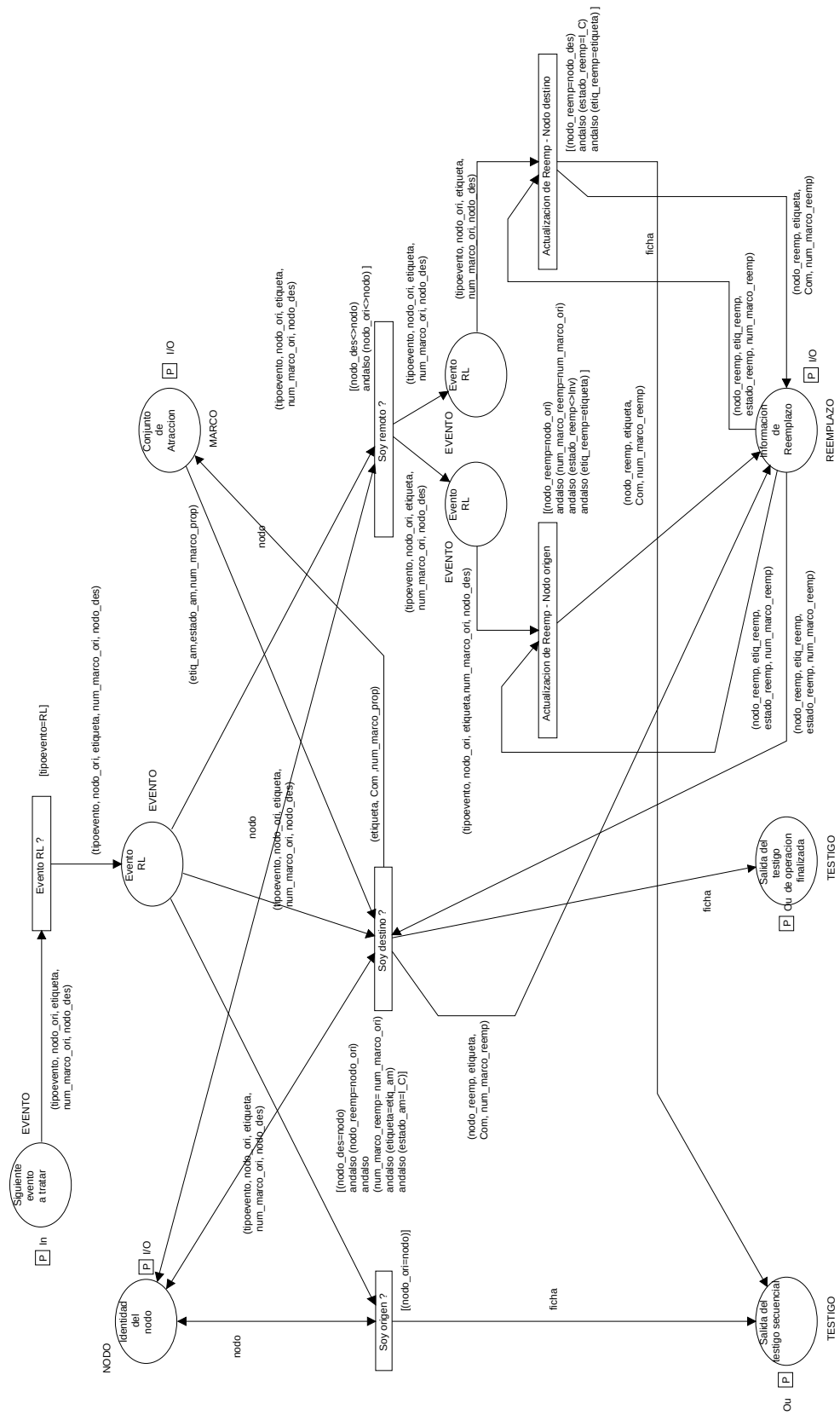


Figura A.24: Página “BusRack”.

A.7.3 Página "BusWreq"

La página "BusWreq", que aparece en la figura A.25, modela el comportamiento del controlador de coherencia cuando se trata de procesar un evento entrante del tipo BusWreq.

Tras recibir este evento, los casos posibles son los descritos en la sección 4.9.4:

- El nodo local es origen del evento.
- El nodo local es destino y no propietario.
- El nodo local es remoto.
- El nodo local es destino y propietario pero está pendiente una operación de desalojo.
- El nodo local es destino y propietario pero el bloque está en un estado transitorio.
- El nodo local es destino y propietario.

A.7.4 Página "BusWack"

La página "BusWack", que aparece en la figura A.26, modela el comportamiento del controlador de coherencia cuando se trata de procesar un evento entrante del tipo BusWack.

Tras recibir este evento, los casos posibles son los descritos en la sección 4.9.5:

- El nodo local es origen del evento.
- El nodo local es destino del evento.
- El nodo local es remoto.

Sólo se genera el testigo secuencial en los casos 16 y 17: si el nodo local es destino del evento, se genera en su lugar el testigo de operación finalizada, al objeto de permitir la finalización de la operación de memoria antes de continuar con el tratamiento de los eventos entrantes.

Si el nodo local no es el origen de este evento, la recepción de este evento obliga a invalidar la eventual copia que pueda existir en la memoria de atracción local (lo que se realiza en la página "Invalida AM local"). Además, debe actualizarse la información de reemplazo correspondiente a los demás nodos que posean una copia (lo que se realiza en la página "Invalidación de información de reemplazo").

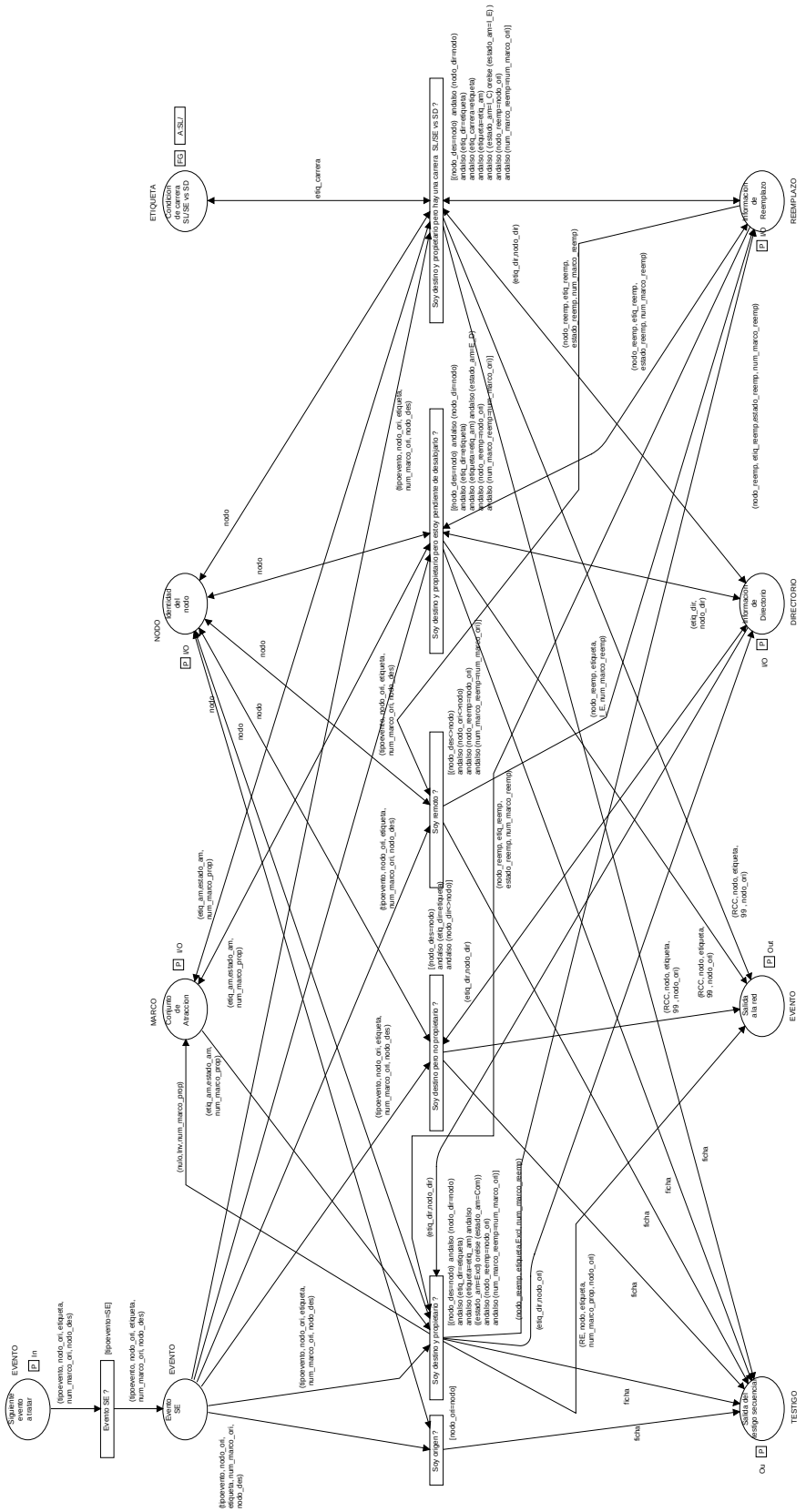


Figura A.25: Página “BusWreq”.

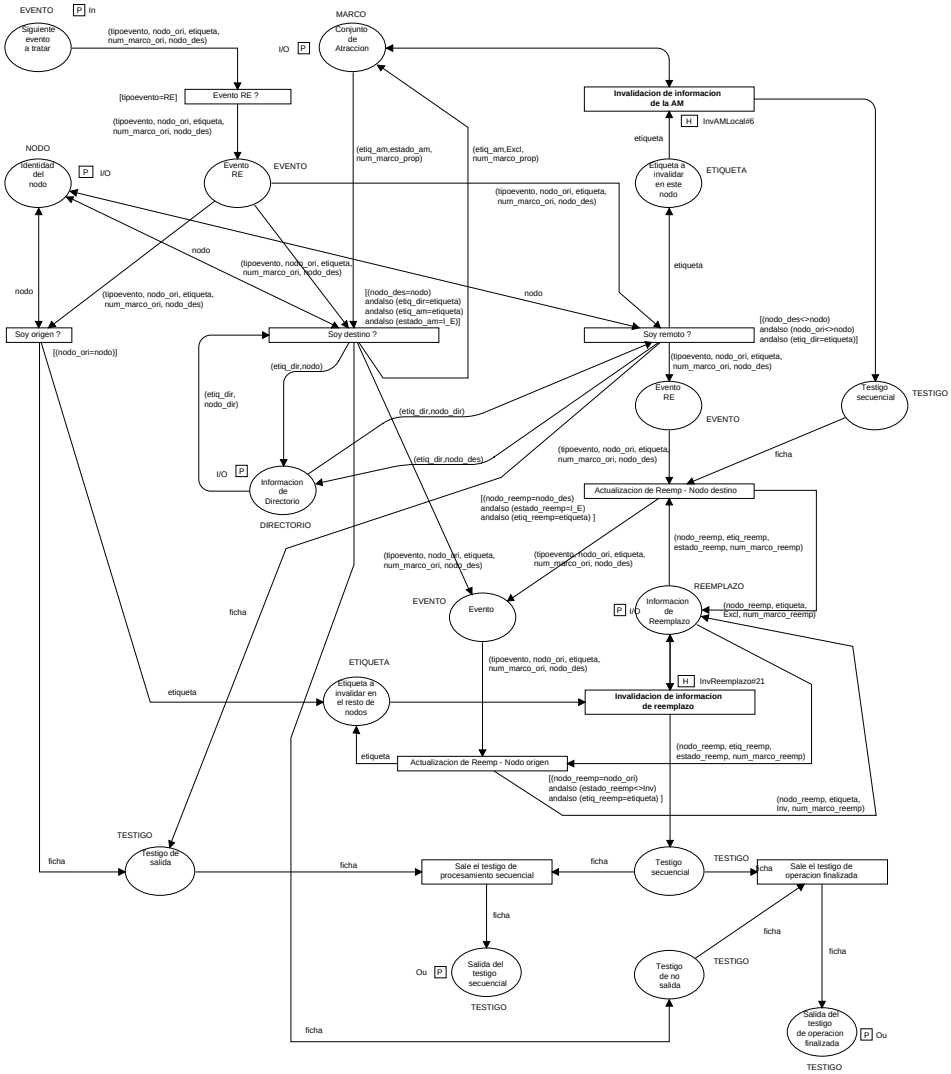


Figura A.26: Página “BusWack”.

A.7.5 Página "BusFInv"

La página "BusFInv", que aparece en la figura A.27, modela el comportamiento del controlador de coherencia cuando se trata de procesar un evento entrante del tipo BusFInv.

Tras recibir este evento, los casos posibles son los descritos en la sección 4.9.6:

- El nodo local es destino del evento.
- El nodo local es remoto.

Si el nodo local no es el origen de este evento, la recepción de este evento obliga a invalidar la eventual copia que pueda existir en la memoria de atracción local (lo que se realiza en la página "Invalida AM local"). Además, debe actualizarse la información de reemplazo correspondiente a los demás nodos que posean una copia (lo que se realiza en la página "Invalidación de información de reemplazo").

A.7.6 Página "Invalidación de reemplazo"

La página "Invalidación de reemplazo", que aparece en la figura A.28, modela el comportamiento del controlador de coherencia cuando se trata de invalidar en la información de reemplazo todas las copias de un bloque concreto presente en otros nodos. Esta página es invocada por transiciones de substitución situadas en las páginas "BusWack" y "BusFInv". Consiste básicamente en un bucle que toma sucesivamente cada una de las entradas que aparecen en la información de directorio y cuyas etiquetas coincidan con la etiqueta a actualizar, y se cambia el estado a "Inválido", devolviendo la entrada a su sitio. El proceso se repite hasta que no queden etiquetas por invalidar.

A.7.7 Página "Invalidación de AM local"

La página "Invalidación de AM local", que aparece en la figura A.29, modela el comportamiento del controlador de coherencia cuando se trata de invalidar en la memoria de atracción local una copia de un bloque concreto. Si existe una entrada de la memoria de atracción con esa etiqueta, se la procesa modificando su estado a "Inválido". Tras ello, se devuelve el testigo para continuar con el procesamiento. Esta página se invoca a través de transiciones de substitución que se encuentran en las páginas "BusWack" y "BusFInv".

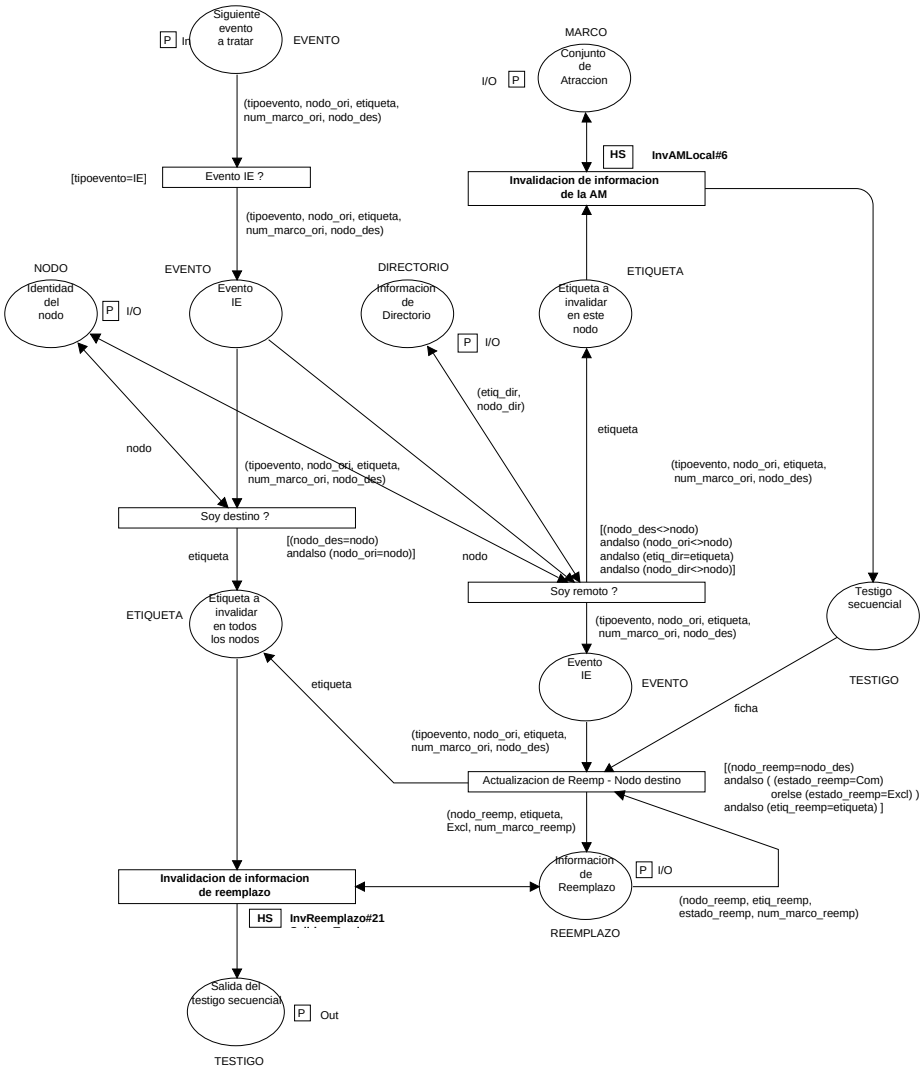


Figura A.27: Página “BusFInv”.

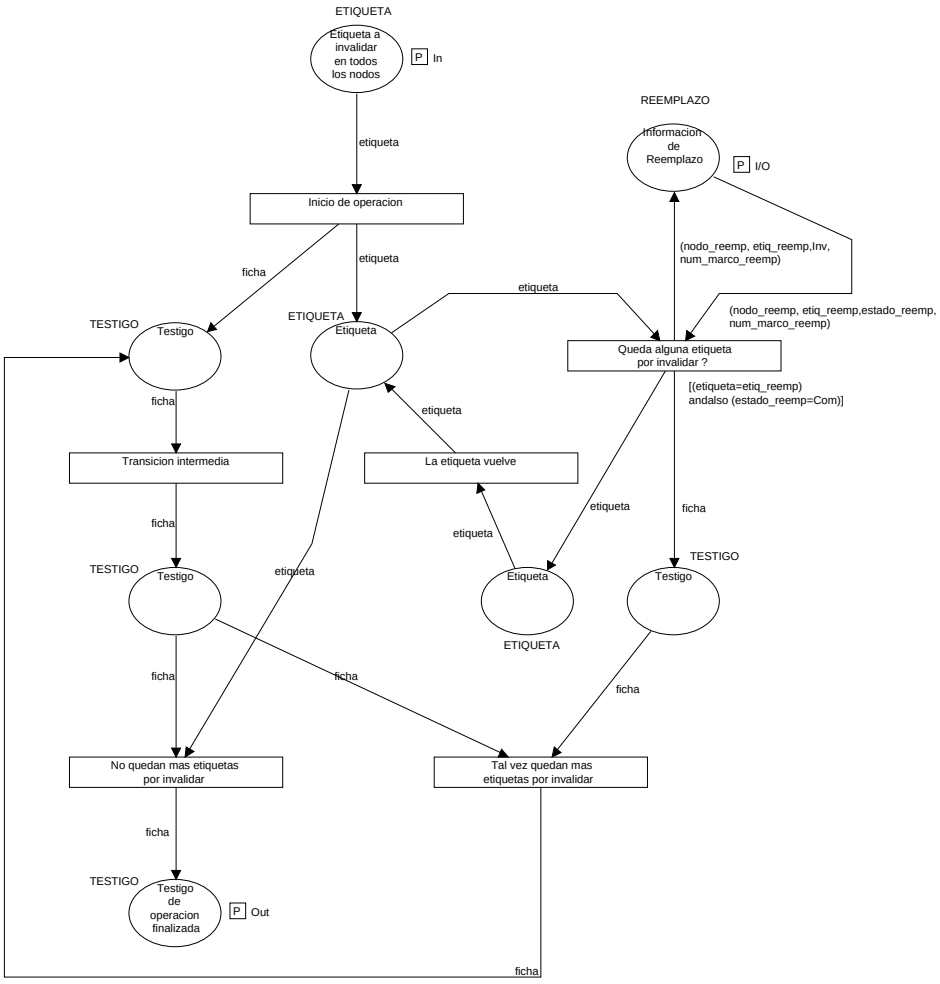


Figura A.28: Página “Invalidación de reemplazo”.

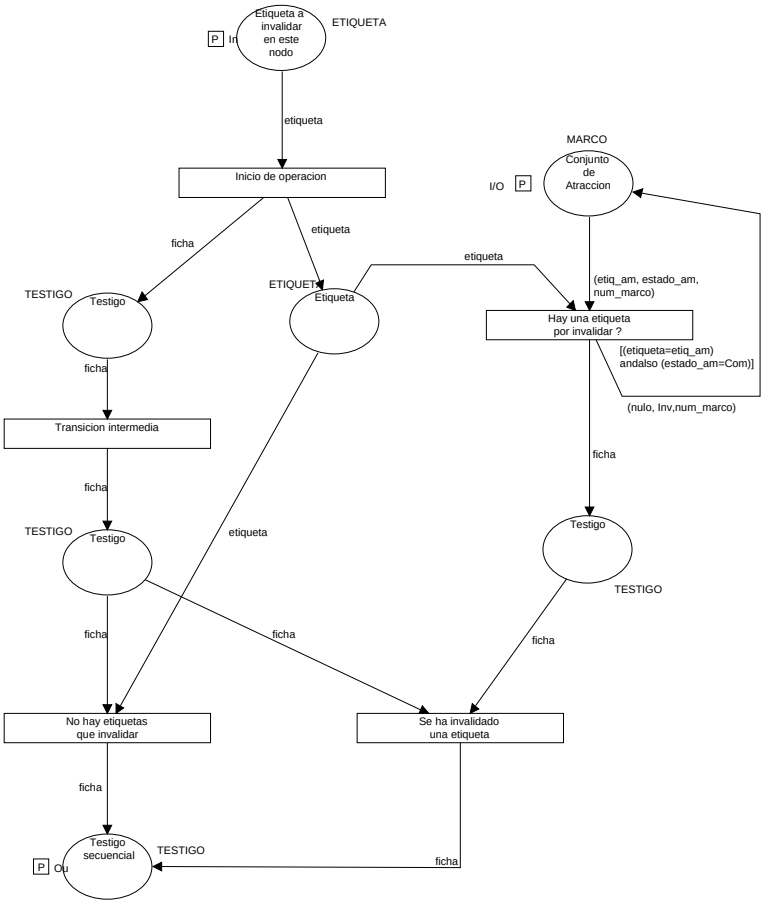


Figura A.29: Página “Invalidación de AM local”.

A.7.8 Página "BusEXreq"

La página "BusEXreq", que aparece en la figura A.30, modela el comportamiento del controlador de coherencia cuando se trata de procesar un evento entrante del tipo BusEXreq.

A.7.9 Página "BusEXreq Destino"

La página "BusEXreq Destino", que aparece en la figura A.31, modela el comportamiento del controlador de coherencia cuando se trata de procesar un evento entrante del tipo BusEXreq y el nodo local es el destino del evento. Ante todo, se comprueba si el nodo local estaba precisamente solicitando esa etiqueta. En ese caso, el bloque se acepta a través de un evento de respuesta de reemplazo, almacenándose en la memoria de atracción local del nodo y activando el testigo que indica que se ha recibido este bloque como producto de un desalojo. Este punto es importante ya que el nodo que inició el desalojo lo hizo antes de recibir la solicitud producida por el nodo local, y cuando la procese responderá con un evento BusRACE.

Si no se estaba solicitando ese bloque, se comprueba si se posee una copia en estado "Compartido" del bloque. En caso contrario, se busca un marco "Inválido" para poder almacenarlo. Si el nodo no posee ningún marco "Inválido", se busca un bloque en estado "Compartido no Propietario", para sobreescribirlo con el bloque recibido. Si no se posee ninguno, el bloque se rechaza a través de un evento de respuesta negativa de desalojo (BusEXnak). El testigo secuencial que aparece en la parte inferior de la figura permitirá seguir procesando los eventos entrantes.

A.7.10 Página "BusEXack"

La página "BusEXack", que aparece en la figura A.32, modela el comportamiento del controlador de coherencia cuando se trata de procesar un evento entrante del tipo BusEXack.

A.7.11 Página "BusEXnak"

La página "BusEXnak", que aparece en la figura A.33, modela el comportamiento del controlador de coherencia cuando se trata de procesar un evento entrante del tipo BusEXnak.

A.7.12 Página "BusRACE"

La página "BusRACE", que aparece en la figura A.34, modela el comportamiento del controlador de coherencia cuando se trata de procesar un evento entrante del tipo BusRACE. Como se ha descrito en la sección 4.9.10, este evento no implica la modificación de la información de la memoria de

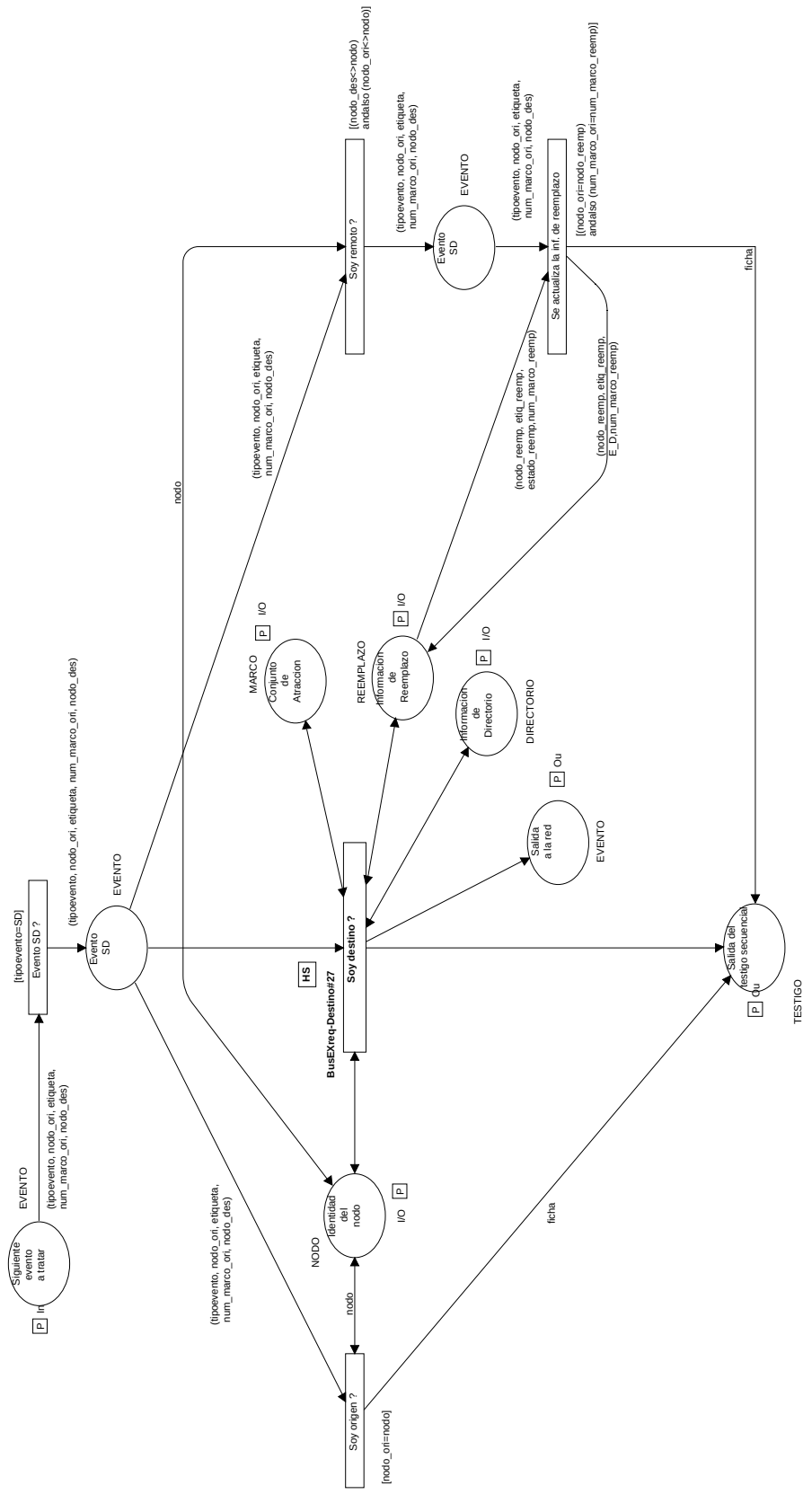


Figura A.30: Página “BusEXreq”.

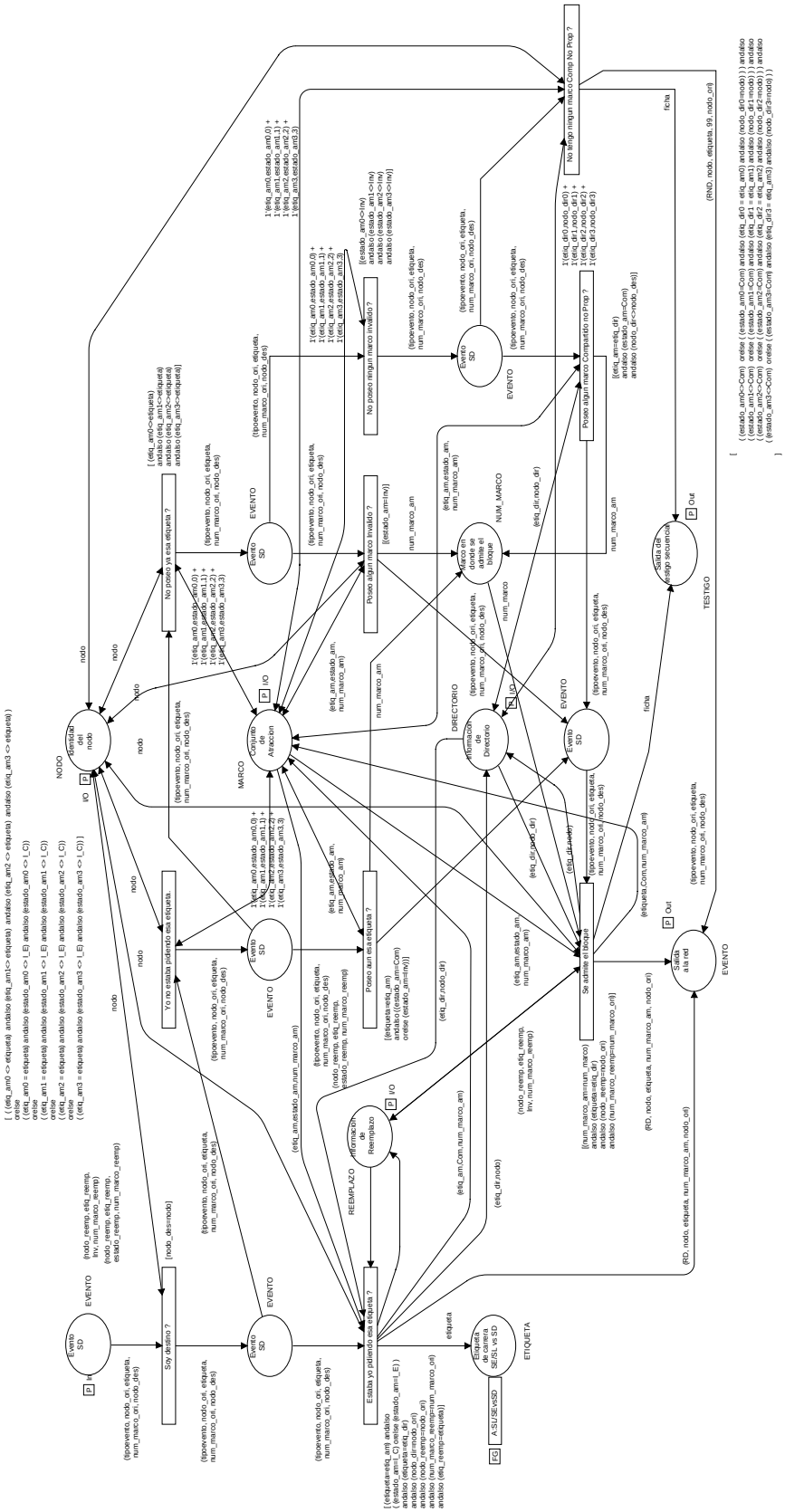


Figura A.31: Página “BusEXreq Destino”.

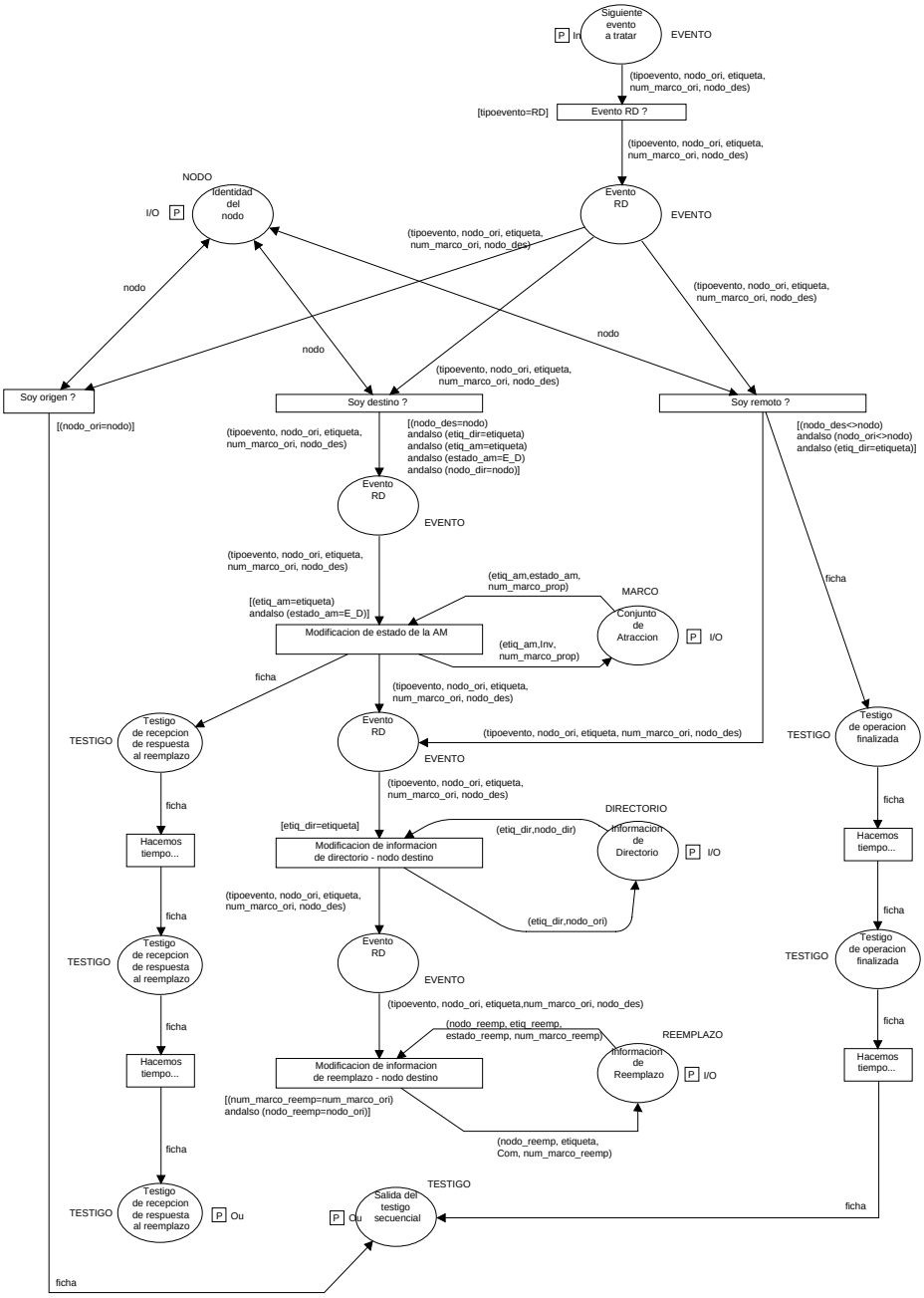


Figura A.32: Página “BusEXack”.

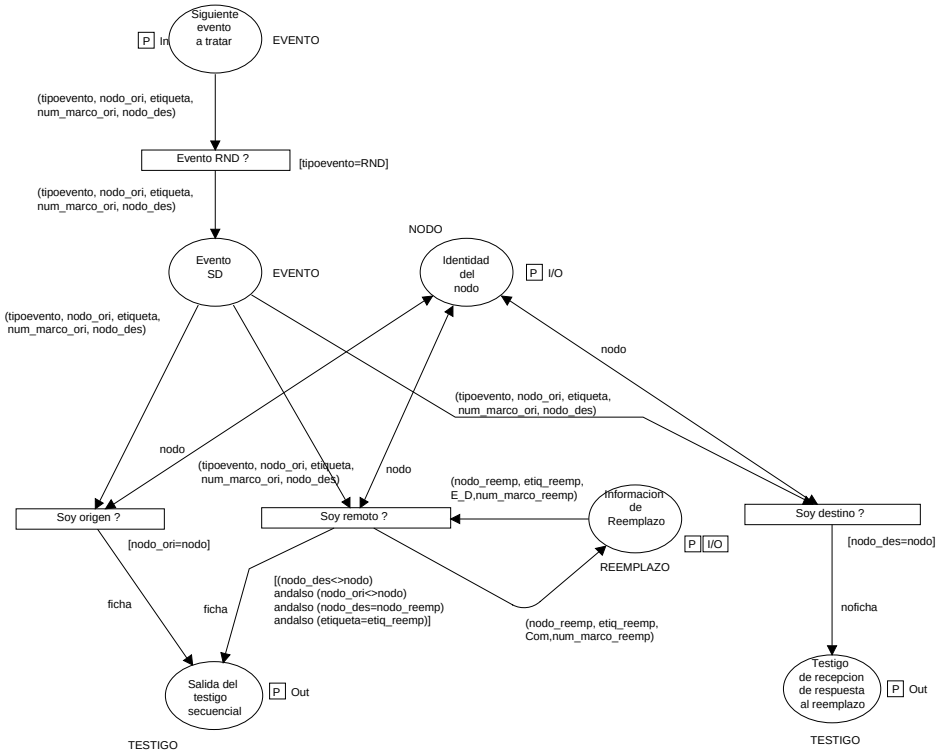


Figura A.33: Página “BusEXnak”.

atracción, la de reemplazo o el directorio. Por lo tanto, en el modelo de VSR-COMA realizado con redes de Petri sólo se comprueba si el nodo local es el destinatario del evento y si hay alguna operación pendiente con esa etiqueta. En ese caso se genera un testigo RCC, que será procesado por la página que generó la solicitud.

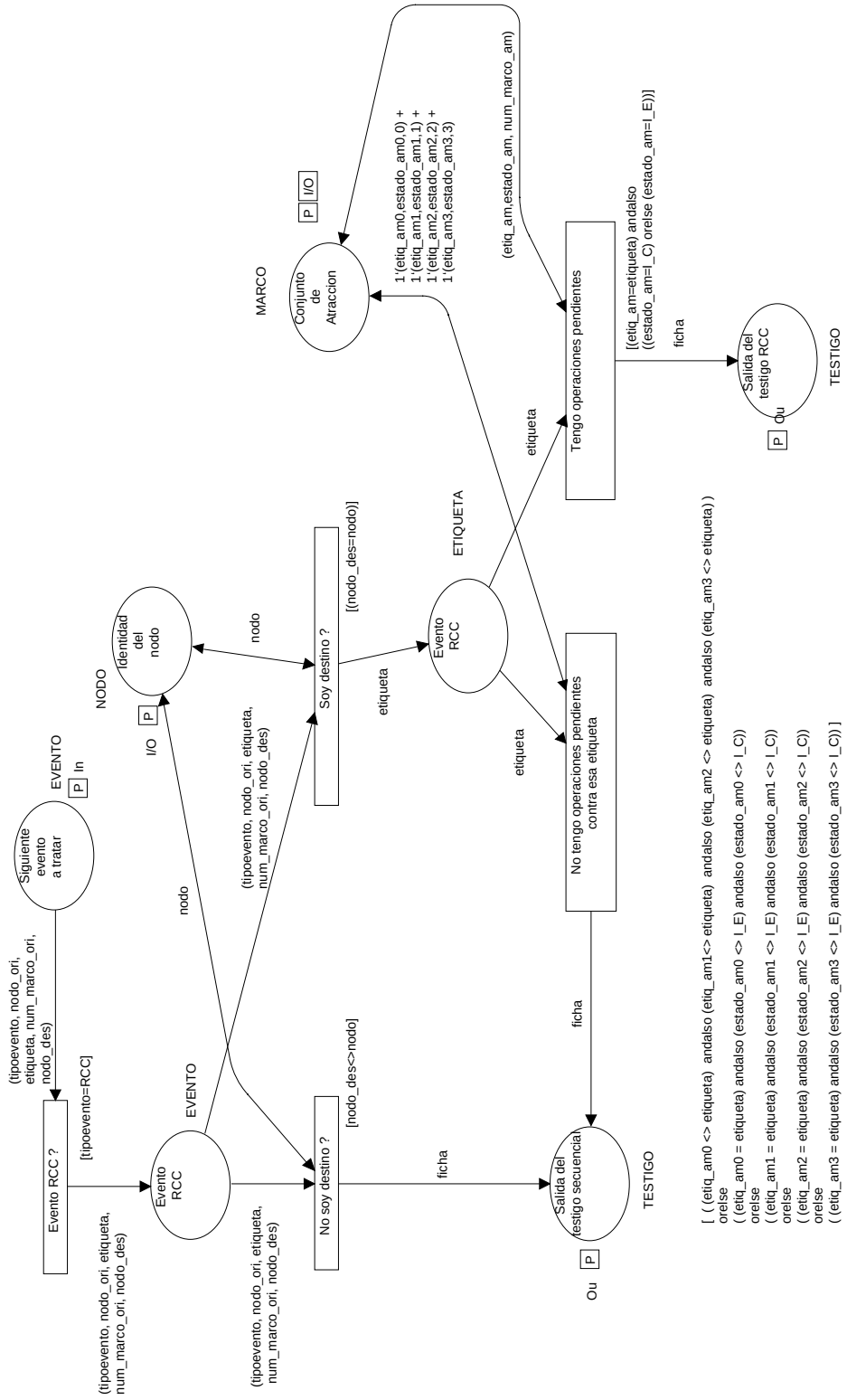


Figura A.34: Página “BusRACE”.

Bibliografía

[1] Sarita V. Adve and Kourush Gharachorloo. Shared memory consistency models: a tutorial. *IEEE Computer*, pages 66–76, December 1996.

[2] Anant Agarwal, Ricardo Bianchini, David Chaiken, Kirk L. Johnson, David Kranz, John Kubiatowics, Beng-Hong Lim, Kenneth Mackenzie, and Donald Yeung. The MIT Alewife machine: architecture and performance. In *Proceedings of the 22nd International Symposium on Computer Architecture (ISCA)*, 1995.

[3] Thomas E. Anderson, David E. Culler, and David A. Patterson. A case for NOW (networks of workstations). *IEEE Micro*, pages 54–64, February 1995.

[4] Sujoy Basu and Josep Torrellas. Enhancing memory use in Simple COMA: Multiplexed Simple COMA. In *4th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 152–161, February 1998.

[5] John K. Bennett, John B. Carter, and Willy Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Second ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 168–176, March 1990.

[6] Guy E. Blelloch, Charles E. Leiserson, Bruce M. Maggs, C. Greg Plaxton, Stephen J. Smith, and Marco Zagha. A comparison of sorting algorithms for the connection machine CM-2. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 3–16, July 1991.

[7] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, JakovÑ. Seizovic, and Wen-King Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, pages 29–36, February 1995.

[8] William J. Bolosky, Robert P. Fitzgerald, and Michael L. Scott. Simple but effective techniques for NUMA memory management. In *Pro-*

ceedings of the Twelfth ACM symposium on Operating systems principles, pages 19–31, December 1989.

[9] William J. Bolosky, Michael L. Scott, Robert P. Fitzgerald, Robert J. Fowler, and Alan J. Cox. NUMA policies and their relation to memory architecture. In *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, pages 212–221, April 1991.

[10] F. Burns, A. Koelmans, and A. Yakovlev. Modelling of Superscalar Processor Architectures with Design / CPN. In *CPN'98 Workshop, Aarhus, Denmark*, June 1998.

[11] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: Characterization and metodological considerations. In *Proceedings of the 22nd Anual International Symposium on Computer Architecture*, pages 24–36, June 1995.

[12] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and performance of Munin. In *Proceedings of the thirteenth ACM symposium on Operating systems principles*, pages 152–164, October 1991.

[13] David Chaiken, Craig Fields, Kiyosi Kurihara, and Anant Agarwal. Directory-based cache coherence in large-scale multiprocessors. *Computer*, pages 49–58, June 1990.

[14] Rohit Chandra, Kourosh Gharachorloo, Vijayaraghavan Soundararajan, and Anoop Gupta. Performance evaluation of hybrid hardware and software distributed shared memory protocols. In *Proceedings of the 8th ACM International Conference on Supercomputing (ICS), Manchester, England*, July 1994.

[15] Sangyeun Cho, Jinseok Kong, and Gyungho Lee. Coherence and Replacement Protocol of DICE - A Bus Based COMA Multiproccesor. *Journal of Parallel and Distributed Computing*, pages 14–32, April 1999.

[16] Computer Science Department of University of Aarhus and Meta Software Corporation. *Design/CPN Reference Manual*, 3 edition, 1996. URL=http://www.daimi.au.dk/designCPN.

[17] Alan L. Cox and Robert J. Fowler. The implementation of a coherent memory abstraction on a NUMA multiprocessor: experiences with platinum. In *Proceedings of the Twelfth ACM symposium on Operating systems principles*, pages 32–44, December 1989.

[18] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus ErikSchauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: Towards a realistic model of parallel computation. In *Proceedings of 4th ACM Symposium on Principles and Practice of Parallel Programming*, San Diego, California, 1993.

[19] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus ErikSchauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: A Practical Model of Parallel Computation. *Communications of the ACM*, pages 78–85, November 1996.

[20] David E. Culler and Jaswinder Pal Singh. *Parallel Computer Architecture: A Hardware/Software Approach.* Morgan Kauffman, 1999.

[21] Fredrik Dahlgren and Anders Landin. Reducing the replacement overhead in bus-based COMA multiprocessors. In *Proceeding of the 3rd International Symposium on High-performance Computer Arquitecture (ISCA)*, February 1997.

[22] Fredrik Dahlgren and Josep Torrellas. Cache-only memory architectures. *IEEE Computer*, pages 72–79, June 1999.

[23] Helen Davis and Stephen R. Goldschmidt. Tango: A multiprocessor simulation and tracing system. Technical Report CSL-TR-90-439, Computer System Laboratory, Stanford University, July 1990.

[24] Agustín De Dios Hernández, Benjamín Sahelices Fernández, and Diego R. Llanos Ferraris. Una aproximación a la verificación del protocolo de coherencia cache del sistema COMA-BC. In *Proceedings X Jornadas de Paralelismo, Departamento de Informática, Universidad de Murcia, España*, pages 21–27, Septiembre 1999.

[25] Thomas H. Dunigan. Kendall Square Multiprocessor: Early Experiences and Performance. Technical report, Mathematical Sciences Section, Oak Ridge National Laboratory, October 1994.

[26] Thomas H. Dunigan. Multi-ring performance of the Kendall Square Multiprocessor. Technical report, Mathematical Sciences Section, Oak Ridge National Laboratory, October 1994.

[27] Kourosh Gharachorloo. *Memory consistency models for shared-memory multiprocessors.* PhD thesis, Department of Electrical Engineering, Stanford University, December 1995.

[28] L. Giannini and A. Chien. A software architecture for global address space communication on clusters: Put / get on fast messages. In *Proceedings of the 7th High Performance Distributed Computing (HPDC7) Conference*, 1998.

[29] James R. Goodman. Cache consistency and sequential consistency. Technical report, Computer Sciences Department, University of Wisconsin-Madison, 1990.

[30] Gary Graunke and Shreenkan Thakkar. Synchronization algorithms for shared-memory multiproccesors. *Computer*, pages 60–69, June 1990.

[31] Anoop Gupta, Truman Joe, and Per Stenström. Performance limitations of cache-coherent NUMA and hierarchical COMA multiprocessors and the Flat-COMA solution. Technical Report CLS-TR-92-524, Computer Systems Laboratory, Stanford University, May 1992.

[32] Eric Hagersten. *Toward Scalable Cache Only Memory Architectures*. PhD thesis, Department of Telecommunication and Computer Science, Stockholm, Sweden, 1993.

[33] Erik Hagersten, Pär Andersson, Anders Landin, and Seif Haridi. A performance Study of the DDM - A Cache-Only Memory Architecture. Technical report, Swedish Institute of Computer Science, November 1991.

[34] Erik Hagersten, Anders Landin, and Seif Haridi. Moving the shared memory closer to the processors - DDM. Technical Report R90-17B, Swedish Institute of Computer Science, May 1991.

[35] Erik Hagersten, Anders Landin, and Seif Haridi. DDM - A Cache-Only Memory Architecture. *IEEE Computer*, pages 44–54, September 1992.

[36] John Heinlein, Robert P. Bosch, Kourosh Gharachorloo, Mendel Rosenbum, and Anoop Gupta. Coherent block data transfer in the FLASH multiprocessor. In *Proceedings of the 11th International Parallel Processing Symposium (IPPS 97)*, April 1997.

[37] Mark D. Hill. Multiprocessors should support simple memory-consistency models. *IEEE Computer*, pages 28–34, August 1998.

[38] Richard C. Ho, C. Han Yang, Mark A. Horowitz, and David L. Dill. Architecture Validation for Processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, June 1995.

[39] Chris Holt, Mark Heinrich, Jaswinder Pal Singh, Edward Rothberg, and John Hennessy. The effecs of latency, occupancy and bandwidth in distributed shared memory multiprocessors. Technical Report CSL-TR-95-660, Stanford University, January 1995.

[40] Juan A. Illescas, Benjamín Sahelices Fernández, and Agustín De Dios Hernández. Pegaxo, un entorno de simulación de sistema multiprocesador de memoria compartida. In *Proceedings VIII Jornadas de Paralelismo, Departamento de Informática, Universidad de Extremadura, España*, 1997.

[41] Sujat Jamil and Gyungho Lee. Unallocated memory space in coma multiprocessors. In *Proceeding of the 8th International Conference on Parallel and Distributed Computing and Systems*, September 1995.

[42] Kurt Jensen. An Introduction to the Theoretical Aspects of Coloured Petri Nets. In *A Decade of Concurrency, Lecture Notes in Computer Science*, volume 803, pages 230–272. Springer-Verlag, 1994.

[43] Kurt Jensen. A brief introduction to Coloured Petri Nets. In *Proceeding of the TACAS'97 Workshop, Enschede, The Netherlands*, 1997.

[44] Kurt Jensen. An Introduction to the Practical use of Coloured Petri Nets. In *Lectures on Petri Nets II: Applications, Lecture Notes in Computer Science*, pages 237–292. Springer-Verlag, 1998.

[45] Truman Joe. *COMA-F: A Non-hierarchical Cache Only Memory Architecture*. PhD thesis, Department of Electrical Engineering, Stanford University, 1995.

[46] Truman Joe and John L Hennessy. Evaluating the memory overhead required for COMA architectures. In *Proceeding of the 21th Annual International Symposium on Computer Architecture*, April 1994.

[47] K.K. Keeton, T.E. Anderson, and D.A. Patterson. LogP quantified: The case for low overhead local area networks. In *Proceedings of Hot Interconnects III: A Symposium on High Performance Interconnects*, Stanford, California, 1995. Stanford University.

[48] Brian Kernighan and Rob Pike. *El entorno de programación Unix*. Prentice-Hall, 1987.

[49] Brian Kernighan and Dennis Ritchie. *El lenguaje de programación C*. Prentice-Hall, 2 edition, 1991.

[50] Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John Hennessy. The Stanford FLASH multiprocessor. In *Proceedings of the 21st International Symposium on Computer Architecture (ISCA)*, pages 302–313, April 1994.

[51] Anders Landin and Fredrik Dahlgren. Bus-based COMA - Reducing Traffic in Shared-Bus Multiprocessors. In *Second IEEE Conference on High Performance Computer Architectures*, February 1996.

[52] Anders Landin and Mattias Karlgren. A Study of the Efficiency of Shared Attraction Memories in Cluster-Based COMA Multiprocessors. In *Eleventh International Parallel Processing Symposium*, April 1997.

[53] Gyungho Lee. An assessment of COMA multiprocessors. Technical report, Department of Electrical Engineering, University of Minnesota, 1995.

[54] Gyungho Lee and Sujat Jamil. Memory block relocation in cache-only memory multiprocessors. Technical report, Department of Electrical Engineering, University of Minnesota, 1995.

[55] Gyungho Lee and Jinseok Kong. Prospects of distributed shared memory for reducing global traffic in shared-bus multiprocessors. Technical report, Department of Electrical Engineering, University of Minnesota, 1995.

[56] Gyungho Lee, Bland QuattleBaum, Sangyeun Cho, and Larry Kinney. Global Bus Design of a Bus-Based COMA Multiprocessor DICE. In *Proceedings of International Conference on Computer Design*, October 1996.

[57] Gyungho Lee, Bland Quattlebaum, and Larry Kinney. Protocol mapping for a bus-based COMA multiprocessor. Technical report, Department of Electrical Engineering, University of Minnesota, 1994.

[58] Daniel Lenoski, James Laudon, and Kourosh Gharachorloo. The Stanford DASH Multiprocessor. *IEEE Computer*, pages 63–79, March 1992.

[59] Daniel Lenoski, James Laudon, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta, and John Hennessy. The DASH prototype: implementation and performance. *IEEE Transactions on Parallel and Distributed Systems*, January 1993.

[60] Daniel Lenoski, James Laudon, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta, and John Hennessy. The DASH prototype: logic overhead and performance. *IEEE Transactions on parallel and distributed systems*, January 1993.

[61] K. Li and P Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, pages 321–359, November 1989.

[62] Diego R. Llanos Ferraris. *Curso de C bajo Unix*. Secretariado de Publicaciones de la Universidad de Valladolid, 1998.

[63] Diego R. Llanos Ferraris, Benjamín Sahelices Fernández, and Agustín De Dios Hernández. Diseño de un protocolo COMA en bus común distribuido con reemplazo. In *Proceedings X Jornadas de Paralelismo, Departamento de Informática, Universidad de Murcia, España*, pages 241–246, Septiembre 1999.

[64] Tom Lovett and Russell Clapp. STiNG: a CC-NUMA computer system for the commercial marketplace. In *Proceedings of the 23rd annual international symposium on Computer architecture (ISCA)*, pages 308–317, May 1996.

[65] E. Lusk, R. Overbeek, J. Boyle, R. Butler, T. Disz, B. Glickfeld, J. Patterson, and R. Stevens. *Portable Programs for Parallel Processors*. Holt, Rinehart and Winston, Inc., 1987.

[66] Peter Magnusson, Anders Landin, and Erik Hagersten. Efficient software synchronization on large cache coherent multiprocessors. Technical report, Swedish Institute of Computer Science, 1994.

[67] R. Martin, A.M. Vahdat, E. Culler, and E. Anderson. Effects of communication latency, overhead and bandwith in a cluster architecture. In *Proceedings of 24th Int. Symposium on Computer Architecture, ISCA*, 1997.

[68] José F. Martínez, Josep Torrellas, and José Duato. Improving the performance of bristled CC-NUMA systems using virtual channels and adaptivity. In *Proceedings of the 1999 international conference on Supercomputing*, pages 202–209, June 1999.

[69] Oliver McBryan. KSR1 Computer. Technical report, University of Colorado, 1992.

[70] Maged M. Michael, Ashwini K. Nanda, Beng-Hong Lim, and Michael L. Scott. Coherence controller architectures for SMP-based CC-NUMA multiprocessors. In *Proceedings of the 24th international symposium on Computer architecture (ISCA)*, pages 219–228, June 1997.

[71] Andreas Moestedt. The DDMlite, design and implementation of a COMA multiprocessor. Technical report, Swedish Institute of Computer Science, Lund University, December 1995.

[72] Farnaz Mounes-Toussi and David J. Lilja. Reducing the impact of false-sharing using a write-through cache with partial block invalidation. Technical Report HPPC-94-15, Department of Electrical Engineering, University of Minnesota, December 1994.

[73] Farnaz Mounes-Toussi and David J. Lilja. The effect of using state-based priority information in a shared-memory multiprocessor cache replacement policy. In *Proceeding of the International Conference on Parallel Processing, Minneapolis, MN*, August 1998.

[74] Bill Nitzberg and Virginia Lo. Distributed shared memory: A survey of issues and algorithms. *Computer*, pages 52–60, August 1991.

[75] S. Pakin, M. Lauria, and A. Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet. In *Proceedings of Supercomputing'95*, 1995.

[76] Jaswinder Pal Singh, Anoop Gupta, and Marc Levoy. Parallel visualization algorithms: Performance and architectural implications. *IEEE Computer*, pages 45–55, July 1994.

[77] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. Splash, Stanford Parallel Applications for Shared Memory. *Computer Architecture News*, pages 5–44, 1992.

[78] C. A. Petri. *Kommunikation mit Automaten*. PhD thesis, Universidad de Bonn, Alemania, 1962.

[79] Sanjai Raina. *Emulation of a Virtual Shared Memory Architecture*. PhD thesis, University of Bristol, UK, 1993.

[80] Jean-Marie Rifflet. *Comunicaciones en Unix*. McGraw Hill, 1990.

[81] Kay A. Robbins and Steven Robbins. *Unix programación práctica*. Prentice-Hall, 1997.

[82] E. Rosti, E. Smirni, T. D. Wagner, A. W. Apon, and L. W. Dowdy. The KSR1: Experimentation and Modeling of Poststore. Technical report, Department of Computer Science, Vanderbilt University, Nashville, 1993.

[83] Edward Rothberg, Jaswinder Pal Singh, and Anoop Gupta. Working sets, cache sizes, and node granularity issues for large-scale multiprocessors. In *Proceedings of the 20th annual symposium on computer architecture*, pages 14–23, May 1993.

[84] Benjamín Sahelices Fernández. *COMA-BC: una Arquitectura de Memoria Sólo Cache en Bus Común no Jerárquica*. PhD thesis, Departamento de Informática, Universidad de Valladolid, España, 1998.

[85] Benjamín Sahelices Fernández, Agustín De Dios Hernández, and J. J. Cabana. A COMA multicomputer system for one-line common bus. In *Proceedings of the 1st IASTED International Conference on Parallel and Distributed Systems, Euro-PDS'97*, pages 38–41, 1997.

[86] Benjamín Sahelices Fernández, Juan Illescas, and Luis Alonso Romero. COMA-BC: A cache only memory architecture multicomputer for non-hierarchical common bus networks. In *Proceedings of the 6th Euromicro Workshop on Parallel and Distributed Processing*, pages 502–508, 1998.

[87] Benjamín Sahelices Fernández, Juan A. Illescas, and Agustín De Dios Hernández. COMA-BC: una arquitectura de memoria sólo cache en bus común no jerárquica. In *Proceedings VIII Jornadas de Paralelismo, Departamento de Informática, Universidad de Extremadura, España*, pages 183–192, 1997.

[88] Ashley Saulsbury, Anders Landin, and Erik Hagersten. COMA machines can be easily built. In *International Symposium on Computer Architecture, Shared Memory Workshop*, 1994.

[89] Ashley Saulsbury and Andreas Nowatzyk. Simple Coma on S3.mp. In *Proceedings of the 1995 International Symposium on Computer Architecture (Shared Memory Workshop)*, June 1995.

[90] Ashley Saulsbury, Tim Wilkinson, John Carter, and Anders Landin. An argument for Simple COMA. In *First IEEE Symposium on High Performance Computer Architecture (HPCA)*, pages 276–285, January 1995.

[91] Richard Thomas Simoni. *Cache coherence directories for scalable multiprocessors*. PhD thesis, Department of Electrical Engineering, Stanford University, March 1995.

[92] Jaswinder Pal Singh, John L. Hennessy, and Anoop Gupta. Implications of hierarchical N-body methods for multiprocessor architectures. *ACM Transactions on Computer Systems*, 13(2):141–202, 1995.

[93] Per Stenström, Erik Hagersten, David J. Lilja, Margaret Martonosi, and Madan Venugopal. Trends in shared memory multiprocessing. *Computer*, pages 44–50, December 1997.

[94] Per Stenström, Truman Joe, and Anoop Gupta. Comparative performance evaluation of cache-coherent NUMA and COMA architectures. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, May 1992.

[95] Peter Stenström. A survey of cache coherence schemes for multiprocessors. *Computer*, pages 12–24, June 1990.

[96] W. Richard Stevens. *Advanced programming in the Unix environment*. Addison-Wesley, 1993.

[97] Josep Torrellas. Perfomance and implementation issues in shared-memory multiprocessors. Technical report, Center for Supercomputing Research & Development, University of Illinois at Urbana-Champaign, September 1996.

[98] Josep Torrellas and David Padua. The Illinois Aggresive COMA Multiprocessor Project. In *Sixth Symposium on the Frontiers of Massively Parallel Computing*, October 1996.

[99] Ben Verghese, Scott Devine, Anoop Gupta, and Mendel Rosenblum. Operating system support for improving data locality on CC-NUMA Computer Servers. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 279–289, 1996.

[100] Mary K. Vernon and Mark A. Holliday. Performance analysis of multiprocessor cache consistency protocols using generalized timed Petri nets. In *Proceedings of the joint conference on computer performance modelling, measurement and evaluation*, pages 9–17, May 1986.

[101] Steve Cameron Woo, Jaswinder Pal Singh, and John Hennessy. The performance advantages of integrating message passing in cache-coherent multiprocessors. Technical Report CSL-TR-93-593, Stanford University, December 1993.

[102] Steve Cameron Woo, Jaswinder Pal Singh, and John L. Hennessy. The performance advantages of integrating block data transfer in cache-coherent multiprocessors. In *Proceedings sixth international conference on Architectural support for programming languages and operating systems*, pages 219–229, October 1994.

[103] David A. Wood and Mark D. Hill. Cost-effective parallel computing. *IEEE Computer*, pages 69–72, February 1995.

[104] Liuxi Yang, Anthony-Trung Nguyen, and Josep Torrellas. How processor-memory integration affects the design of DSM. In *Workshop on Mixing Logic and DRAM: Chips that Compute and Remember*, June 1997.

[105] Liuxi Yang and Josep Torrellas. Speeding up the memory hierarchy in Flat COMA multiprocessors. In *3rd International Symposium on High-Performance Computer Architecture (HPCA)*, January 1997.

[106] Marco Zagha and Guy E. Blelloch. Radix sort for vector multiprocessors. In *Proceedings of the 1991 conference on Supercomputing*, pages 712–722, November 1991.

[107] Z. Zhang and J. Torrellas. Reducing remote conflict misses: NUMA with remote cache versus COMA. In *Proceedings 3rd International Symposium on High Performance Computer Architecture (HPCA)*, pages 272–281, 1997.

[108] W.M. Zuberek, R. Govindarajan, and F. Suciu. Timed Colored Petri Net Models of Distributed Memory Multithreaded Multiprocessors. In *CPN'98 Workshop, Aarhus, Denmark*, June 1998.