Universidad Zaragoza

Trabajo de Fin de Grado

Grado en Ingeniería Informática

# Diseño y evaluación de las memorias cache para un chip multicore alimentado a muy baja tensión.

## Design and evaluation of cache memories for a chip multicore powered at very low voltage


Autor/es

Carlos Escuín Blasco

Agustín Navarro Torres

Director

Pablo Ibáñez Marín

ESCUELA DE INGENIERÍA Y ARQUITECTURA

Universidad de Zaragoza



Septiembre 2016

TRABAJOS DE FIN DE GRADO / FIN DE MÁSTER

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

# DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD

(Este documento debe acompañar al Trabajo Fin de Grado (TFG)/Trabajo Fin de Máster (TFM) cuando sea depositado para su evaluación).

D./Dª. CARLOS ESCUÍN BLASCO,

con nº de DNI 73003863 - T en aplicación de lo dispuesto en el art. 14 (Derechos de autor) del Acuerdo de 11 de septiembre de 2014, del Consejo de Gobierno, por el que se aprueba el Reglamento de los TFG y TFM de la Universidad de Zaragoza,

Declaro que el presente Trabajo de Fin de (Grado/Máster) GRADO, (Título del Trabajo) DISEÑO Y EVALUACIÓN DE LAS MEMORIAS CACHE PARA UN CHIP MULTICORE ALIMENTADO A MUY BAJA TENSIÓN.

es de mi autoría y es original, no habiéndose utilizado fuente sin ser citada debidamente.

Zaragoza, 2 SEPTIEMBRE 2016

Fdo: CARLOS ESCUÍN BLASCO

TRABAJOS DE FIN DE GRADO / FIN DE MÁSTER

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

# DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD

*(Este documento debe acompañar al Trabajo Fin de Grado (TFG)/Trabajo Fin de Máster (TFM) cuando sea depositado para su evaluación).*

D./Dª. AGUSTIN NAVARRO TORRES,

con nº de DNI 25199418C en aplicación de lo dispuesto en el art. 14 (Derechos de autor) del Acuerdo de 11 de septiembre de 2014, del Consejo de Gobierno, por el que se aprueba el Reglamento de los TFG y TFM de la Universidad de Zaragoza,

Declaro que el presente Trabajo de Fin de (Grado/Máster) GRADO, (Título del Trabajo) DISEÑO Y EVALUACIÓN DE LAS MEJORAS COCHE PARA UN CHIP MULTICORE ALIMENTADO A MUY BAJA TENSIÓN,

es de mi autoría y es original, no habiéndose utilizado fuente sin ser citada debidamente.

Zaragoza, 2 SEPTIEMBRE 2016

Fdo: AGUSTIN NAVARRO TORRES

# Diseño y evaluación de las memorias caches para un chip multicore alimentado a muy baja tensión

## Resumen

El ahorro energético es un objetivo de primer orden en la investigación para el desarrollo de nuevos procesadores. La disminución de la tensión de alimentación consigue este objetivo pero, llegado a un límite se producen errores en las celdas de bit. El primer componente que falla es la cache de último nivel (LLC). Existen numerosas propuestas para mitigar la pérdida de prestaciones consecuencia de estos errores.

Estas propuestas van desde la construcción de las celdas de forma robusta para evitar que se produzcan los fallos hasta complejas soluciones a nivel de arquitectura para mitigar el efecto que tienen estos fallos en el rendimiento final del procesador. Existen diversas soluciones arquitecturales: simples como *block disabling* que deshabilita los recursos defectuosos y más complejas como *bdot* que requieren modificar el protocolo de coherencia. Además, todas estas técnicas pueden verse optimizadas incluyendo políticas de reemplazo y modificaciones en la estructura de las memorias caches.

En este trabajo se realizará un estudio sobre técnicas para mejorar el funcionamiento de la LLC a bajo voltaje y, posteriormente, se analizará una nueva propuesta de investigación consistente en utilizar una organización desacoplada para los almacenes de etiquetas y datos. Para ello, se utilizará un simulador detallado a nivel de ciclo basado en el entorno Gem5 de la Universidad de Michigan. Se modelará la nueva propuesta y otras del estado del arte sobre un sistema multiprocesador en chip con una jerarquía de memoria formada por caches privadas y una LLC compartida entre los procesadores. Además, se crearán cargas de trabajo monoprocesador y multiprocesador basadas en SPEC-2k6 y se usarán para realizar la comparación entre las propuestas en términos de tasa de fallos e instrucciones ejecutadas por unidad de tiempo.

# Índice

# 1 Introducción

El consumo eléctrico es uno de los aspectos claves en el diseño de procesadores modernos, ya sea en procesadores móviles para aumentar la duración de la batería de los *smartphones* o en un gran centro de datos para disminuir la factura eléctrica. Por ello, el ahorro energético es un objetivo de primer orden en la investigación para el desarrollo de nuevos procesadores. La disminución de la tensión de alimentación consigue este objetivo pero, llegados a un límite, se producen errores en las celdas de bit. El primer componente que falla es la cache de último nivel (LLC). Existen muchas propuestas para mitigar la pérdida de prestaciones consecuencia de estos errores. En este trabajo se estudia una nueva propuesta de investigación consistente en utilizar una organización desacoplada para los almacenes de etiquetas y datos de la LLC.

Durante años, la industria ha confiado en la reducción de la tensión de alimentación (Vdd) para disminuir el consumo de energía. Sin embargo, la tensión de alimentación está limitada por los ajustados márgenes de operación que tienen las celdas SRAM utilizadas para construir las caches en chip. Por debajo de una tensión umbral, las celdas SRAM no tienen un funcionamiento fiable.

En la literatura se han propuesto distintas soluciones a los problemas que aparecen cuando la cache opera a baja tensión. A nivel electrónico o circuital, puede aumentarse la resilliencia de una celda SRAM incrementando el tamaño de sus transistores o añadiendo circuitería especial [2][3]. Su principal desventaja es que aumentan el área y el consumo del dispositivo.

A nivel microarquitectónico, los diseños de caches tolerantes a fallos se basan en deshabilitar los recursos defectuosos usando distintas granularidades [11]. *Block disabling* (BD) es una de las técnicas estudiadas en este trabajo y consiste en deshabilitar los recursos con granularidad de bloque, es decir, se desconecta toda una entrada de cache cuando en ella se detecta al menos un bit defectuoso.

Otra estrategia (*Block Disabling with Operational Tags, BDOT*), la cual se basa en la técnica *block disabling*, utiliza una fuente natural de redundancia de datos dentro del chip que es la replicación de bloques en la jerarquía de una cache inclusiva. Esta técnica consiste en mantener el vector de etiquetas operativo usando, por ejemplo, celdas robustas para su construcción. De esta manera, se garantiza la inclusión de directorio y una asociatividad constante. Así pues, estas etiquetas, que en la versión de block disabling estarían desactivadas, van a poder ser incluidas en las operaciones de búsqueda y reemplazo.

La principal limitación del esquema BDOT es que no tiene en cuenta el tipo de entrada en la asignación de los bloques a ellas. Si un bloque con un patrón de reuso ha sido asignado a una entrada defectuosa (tipo *T*), todas las peticiones sobre él serán redirigidas a memoria. Por lo tanto, es necesario implementar BDOT junto con una política de gestión de contenidos capaz de distinguir estas situaciones. Esta política identifica los bloques que se van a reusar en el futuro y los guarda en entradas de cache correctas. Su implementación requiere un algoritmo para identificar dichos bloques y un mecanismo para cambiar estos bloques de entradas defectuosas a entradas correctas cuando sea necesario. Además, se necesita un nuevo algoritmo de reemplazo que escoja una de las entradas correctas para realizar el intercambio.

Esta última solución tiene dos problemas: i) se requiere el intercambio de bloques entre entradas defectuosas y entradas correctas, con el consiguiente coste en tiempo y energía. ii) el algoritmo de reemplazo de entrada correcta se limita a su conjunto de cache, lo que restringe mucho sus posibilidades.

En este trabajo se propone el desacoplamiento de los vectores de etiquetas y datos de la LLC para solucionar los problemas expuestos en el párrafo anterior. Una etiqueta podrá asociarse a cualquier entrada del vector de datos mediante un puntero. De esta forma, el intercambio queda reducido a la modificación del puntero correspondiente. Además, el algoritmo de reemplazo de entrada correcta tiene a su disposición todas las entradas del vector de datos, no solo las de conjunto de cache correspondiente a la etiqueta.

## 1.1 Objetivo y alcance

El análisis de nuevas propuestas de investigación en la arquitectura de computadores requiere la utilización de complejos simuladores y herramientas, que permitan imitar el comportamiento de un procesador multicore bajo una carga de trabajo real, para posteriormente analizar sus resultados y poder comprobar o desmentir las hipótesis. La utilización de estas herramientas ocasiona que una nueva persona que desee iniciar una carrera investigadora en este ámbito, como la realización de un máster en investigación o un doctorado, tenga que invertir meses en el aprendizaje de las mismas.

Este proyecto tiene como objetivo afrontar el desafío de aprendizaje de las herramientas anteriormente nombradas: un simulador ampliamente utilizado en el entorno, *Gem5*, y diversas metodologías y herramientas de benchmarking y muestreo como *Simpoints*. Este ejercicio será llevado a cabo mediante el desarrollo de una nueva propuesta de investigación que mejore el ahorro energético de los procesadores a través de su funcionamiento a bajo voltaje.

## 1.2 Metodología y herramientas

Para la validación de nuevas hipótesis y la cuantificación de las mejoras de los nuevos modelos en términos de coste y/o rendimiento, es necesaria la utilización de complejos simuladores. Para la realización de este trabajo se ha optado por la utilización del simulador *Gem5* [3] de la Universidad de Michigan, que permite simular cargas de trabajo reales ciclo a ciclo, como si de un procesador real se tratase.

A la hora de desarrollar el proyecto se trabajó sobre el cluster ATPS del departamento de Informática e Ingeniería de Sistemas, con el fin de usar un entorno de trabajo similar al de un grupo de investigación.

Para la realización de las pruebas se ha optado por la utilización de la suite *SPEC-2K6 [12]*, que reune una serie de *benchmarks* para medir el rendimiento de las CPU, el subsistema de memoria y el compilador. Además, para las pruebas *monocore* se va a utilizar *Simpoint*[6] para el análisis de las aplicaciones y la identificación de las partes representativas de las mismas sobre las que realizar los *checkpoints*. Un checkpoint guarda el estado completo de una máquina en un instante determinado para su posterior ejecución detallada. Estas técnicas permiten tomar estadísticas representativas sin tener que ejecutar en detalle un programa completo, lo que puede llegar a tardar semanas o incluso meses.

Las técnicas de *Simpoint* y *Checkpoint* no están soportadas por *Gem5* para multicore, por consiguiente se utilizará *Fast-forward* en estos casos. *Fast-forward* consiste en la ejecución de manera no detalla de las primeras instrucciones de un programa, sin tenerlas en cuenta para las estadísticas, y, posteriormente, realizar una simulación detallada en la parte representativa que será contabilizada en las estadísticas.

## 1.3 Desglose del trabajo

Debido a la extensión y complejidad del proyecto se optó por su realización entre dos estudiantes. De forma conjunta ambos autores han realizado las siguientes tareas: escritura de la memoria, comprobación cruzada de código, estudio del estado del arte y análisis de los resultados de las simulaciones.

De manera individual, Carlos Escuín Blasco ha realizado: modificación del protocolo de coherencia para que el reemplazo de bloques limpio de caches privadas sea no silencioso, modelado de un nuevo algoritmo de reemplazo específico para cache con entradas defectuosas, creación de cargas y experimentos multiprocesador.

Por otra parte, Agustín Navarro Torres, ha realizado las siguientes tareas de manera individual: implementación de la nueva propuesta BDOT sobre una cache convencional, modelado de la cache con almacenes desacoplados para etiquetas y datos, creación de cargas de trabajo para sistemas monoprocesador mediante *checkpoints* generados a partir de la metodología *simpoint* y los correspondientes experimentos.

## 1.4 Contenido de la memoria

La memoria está estructurada en cuatro secciones: primero se presenta el estado del arte actual de la propuesta, en ella se describe las principales propuestas de la literatura para aumentar la eficacia de las caches funcionando a muy bajo voltaje. En la segunda parte se detalla el modelo implementado, describiendo el modelo base sobre el cual se ha realizado y las principales modificaciones realizadas al mismo. Más adelante, se describe la metodología utilizada para realizar las simulaciones monocore y multicore, además de los resultados obtenidos junto a una breve explicación de los mismo. Por último, hay una breve sección donde se muestra las conclusiones a las que se ha llegado tras la realización del trabajo.

Al final del documento se encuentran las referencias consultadas, y una serie de anexo donde se incluye, una explicación detallada de todos los estados, transiciones y acciones que componen el protocolo de coherencia base, los cambios realizados sobre el protocolo completo para implementar *BDOT*, mostrando los nuevos estados y transiciones creados. Por último se incluye un breve manual con el que poder iniciarse en la utilización del simulador.

# 2 Problema, estado del arte y propuesta

Durante años, la industria ha confiado en la reducción de la tensión de alimentación (Vdd) para disminuir el consumo eléctrico. Desgraciadamente, el escalado de tensión de alimentación está limitado por los ajustados márgenes de operación que tienen las celdas SRAM utilizadas para construir las caches *on-chip*. Las celdas SRAM limitan la reducción del voltaje de alimentación a una tensión umbral mínima ($Vdd_{min}$), por debajo de la cual las celdas SRAM no tienen un funcionamiento fiable. Esta tensión umbral mínima suele determinar la tensión mínima del procesador. Actualmente esta tensión se encuentra en el orden de 0.7-1.0V, cuando se utiliza celdas SRAM convencionales, de seis transistores.

## 2.1 Celdas *SRAM* robustas

A nivel circuital, puede aumentarse la resiliencia de una celda SRAM aumentado el tamaño de sus transistores o añadiendo circuitería especial [2], [3]. Su principal inconveniente, es el aumento de área y consumo del dispositivo provocado por la utilización de esta técnica. Procesadores comerciales, como por ejemplo los de la familia Intel Nehalem, utilizan celdas con circuitería adicional (celdas SRAM 8T) en las caches de primer nivel para hacerlas más resilientes [10].

La utilización de esta técnica para caches de primer nivel de un multiprocesador en chip no supone un aumento importante del área, puesto que la L1 ocupa poca superficie del procesador para no limitar la frecuencia del mismo. Por el contrario, las caches de ultimo nivel (*last-level caches,* LLCs) tienen tamaños y asociatividad mayores, por lo que ocupan gran parte de la superficie útil de un chip. Por ello, la utilización de esta técnica supone un incremento considerable en la superficie total necesaria para el mismo tamaño de memoria cache. Por tanto, para construir las LLCs son preferibles las estructuras SRAM formadas por celdas 6T, ya que son más densas que las formadas por celdas más grandes o con circuitería adicional.

Aunque el consumo eléctrico en condiciones normales es mayor con la utilización de estas técnicas, cuando se disminuye la tensión de alimentación el consumo eléctrico baja de manera no lineal, por lo que la utilización de celdas robustas y menos tensión implica un menor consumo eléctrico que con celdas normales.

## 2.2 Block Disabling (*BD*)

A nivel de arquitectura, la literatura propone diseños de cache tolerantes a fallos basados en la deshabilitación de las celdas defectuosas. *Block Disabling* (BD)[1] consiste en deshabilitar los recursos usando una granularidad de bloque, añadiendo un bit de información a cada una de las entradas para indicar si es defectuosa o no. Utilizar BD como técnica de tolerancia a fallos a tensiones muy bajas da lugar a caches con una asociatividad variable, la cual viene determinada por el número y distribución de los fallos.

Además, en las jerarquías inclusivas, la información de coherencia está integrada en la propia LLC. Para garantizar la propiedad de inclusión, cada vez que la LLC expulsa un bloque se invalidan todas las copias del bloque presentes en las caches privadas. A estos bloques invalidados se les llama víctimas por inclusión [6]. Cuando la asociatividad de la LLC es similar a la asociatividad agregada de las caches privadas, las jerarquías en inclusión no se comportan de manera adecuada [7]. BD agrava este problema ya que la asociatividad del último nivel de cache se ve disminuida debido a las entradas defectuosas.

## 2.3 Block Disabling with Operational Tag (*BDOT*)

BDOT es una técnica para mejorar el rendimiento de memorias caches con vías defectuosas que consiste en mantener el vector de etiquetas operativo, y de esta manera, garantizar la inclusión de directorio y una asociatividad constante.

Desde el punto de vista de la coherencia, la identificación de los bloques que se encuentran en las caches privadas mediante el almacenamiento de su etiqueta en el vector de etiquetas de la LLC es suficiente para garantizar la inclusión de directorio. Esta idea es la base de la técnica conocida como BDOT[1]. Así pues, las etiquetas de las entradas defectuosas, que en *block disabling* estarían deshabilitadas, van a poder ser incluidas en las operaciones de búsqueda y reemplazo.

Para implementar este modelo, se requiere garantizar que no existen defectos en el vector de etiquetas de la LLC. Esto puede conseguirse mediante la construcción de celdas utilizando más transistores o mecanismos de corrección de errores (*error correction codes,* ECCs). La utilización de más transistores para la construcción de celdas incrementan el tamaño de la celda un 33% (asumiendo celdas 8T). Esto tiene un impacto directo en el área del chip utilizada para el vector de etiquetas LLC. Sin embargo, el vector de etiquetas de la LLC ocupa un área relativamente pequeña con respecto al área total de la LLC, por lo que este aumento de área no suele sobrepasar el 2% del tamaño total de la LLC.

Desde el punto de vista de la implementación, solamente se requiere la utilización de un bit de información que indique si la entrada es defectuosa o no (como en BD). Las entradas de la LLC se dividen en dos tipos: entradas tipo *T,* el contenedor de datos está defectuoso, sólo se almacena la etiqueta; y entradas tipo *D* el contenedor de datos no presenta fallos. Además, es necesario adaptar el protocolo de coherencia de forma que una petición a una entrada de tipo *T* sea atendida por las caches privadas o por memoria principal. El reenvío de peticiones fuera del chip implica un mayor tráfico y consumo eléctrico de la misma forma que el reenvío de peticiones a las caches privadas implica un incremento en la utilización de la red *on-chip*.

## 2.4 BDOT. Reemplazo e intercambio

Las políticas de reemplazo para caches convencionales, libres de fallos, asumen que todas las entradas pueden almacenar bloques, lo cual es falso cuando hablamos de caches con entradas deshabilitadas pero con etiquetas operacionales.

La principal limitación del esquema BDOT es que no tiene en cuenta el tipo de entrada a la hora de asignar los bloques. Esto puede conllevar situaciones en las cuales el rendimiento del sistema se vea mermado, por ejemplo, si un bloque que muestra un patrón de reuso ha sido asignado a una entrada defectuosa *T*, todas las peticiones sobre

él serán redirigidas a memoria. Por lo tanto, es necesario la utilización de una política de gestión de contenidos que sea capaz de distinguir entre entradas *T* y *D*.

Los algoritmos de reemplazo basados en reúso han demostrado ser eficientes en caches compartidas de útlimo nivel. La localidad de reúso dice que si un bloque ha sido utilizado por lo menos dos veces, este bloque tenderá a ser reusado en el futuro cercano, además indica que los bloques más recientemente reusados son más útiles que los reusados con anterioridad.

La nueva política de reemplazo tendrá como objetivo asignar entradas *D* a los bloques con mayor probabilidad de ser usados en el futuro. Además la nueva política de reemplazo deberá ser capaz de seleccionar e intercambiar bloques entre entradas *T* y *D*, según su reuso y presencia en caches privadas.

## 2.5 BDOT: Desacoplamiento de etiquetas y datos

En las caches convencionales existe mapeo uno a uno entre etiquetas y datos, es decir, cada entrada del conjunto de etiquetas tiene asociada una entrada del conjunto de datos.

En la literatura se ha propuesto el desacoplamiento de los vectores de etiqueta y datos[13][14] con igual o diferente número de entradas. El desacoplamiento [8] de etiquetas y datos consiste en desvincular las etiquetas y los datos permitiendo que cualquier etiqueta se asocie a cualquier contenedor de datos. Esta modificación requiere un sistema de punteros que relacione cada etiqueta con el contenedor de datos correspondiente.

Cuando utilizamos caches convencionales con BDOT, la capacidad de intercambio de entradas entre bloques está limitada a las entradas de un conjunto. Con la inclusión del desacoplamiento de etiquetas y datos en la memoria cache las posibles entradas para realizar un intercambio de bloques son todas las entradas de la LLC, por consiguiente, la eficiencia de la política de intercambio se ve aumentada considerablemente.

# 3 Modelos implementados

En esta sección presentamos el sistema base sobre el cual se va a desarrollar el resto de modelos y los diversos modelos implementados: *block disabling*, *BDOT*, *BDOT* con reemplazo específico y *BDOT* con desacoplamiento de etiqueta y datos.

## 3.1 Sistema Base

El simulador utilizado para la realización del proyecto (Gem5), permite la utilización de numerosas arquitecturas (x86, ARM, ALPHA), protocolos de memoria (MESI, MOESI, MI) y redes de interconexión para la simulación de cargas de trabajo reales.

Para desarrollo de los modelos nombrados en la sección anterior, se ha utilizado como sistema base un procesador fuera de orden — *Out of Order (OOO)* — de arquitectura *ALPHA* de Gem5 con las siguientes características:

- Simulación del procesador *Alpha 21264*.
- Ejecución de diversos sistemas operativos como por ejemplo Linux 2.4/2.6 o FreeBSD.
- Ejecución de programas sin sistema operativo mediante emulación de las llamadas al mismo.
- *Ruby* como modelo del subsistema de memoria que nos proporciona flexibilidad a la hora de modificar el protocolo de coherencia y los distintos componentes del mismo, así como un mayor realismo en la simulación.
- Protocolo de coherencia MESI.
- Dos niveles de memoria cache — L1 y L2 —.
- Jerarquía de memoria cache inclusiva, todos los bloques de L1 se encuentran en L2.

### 3.1.1 Protocolo de coherencia

El protocolo de coherencia es una de las piezas claves de una jerarquía cache, se ocupa de asegurar la integridad de los datos y la coherencia de ellos frente, por ejemplo, a cambios simultáneos sobre un mismo bloque por parte de dos núcleos diferentes. El protocolo base elegido para desarrollar el proyecto es el MESI (Modificado, Exclusivo, Compartido, Inválido) de dos niveles.

**Nivel L1**

El primer nivel de cache (L1) es privado a cada cada uno de los núcleos, y se encuentra dividido en instrucciones y datos. A continuación se expone el diagrama de estados estables y sus transiciones, ver fig 1. Una descripción completa puede observarse en el anexo 1.1.

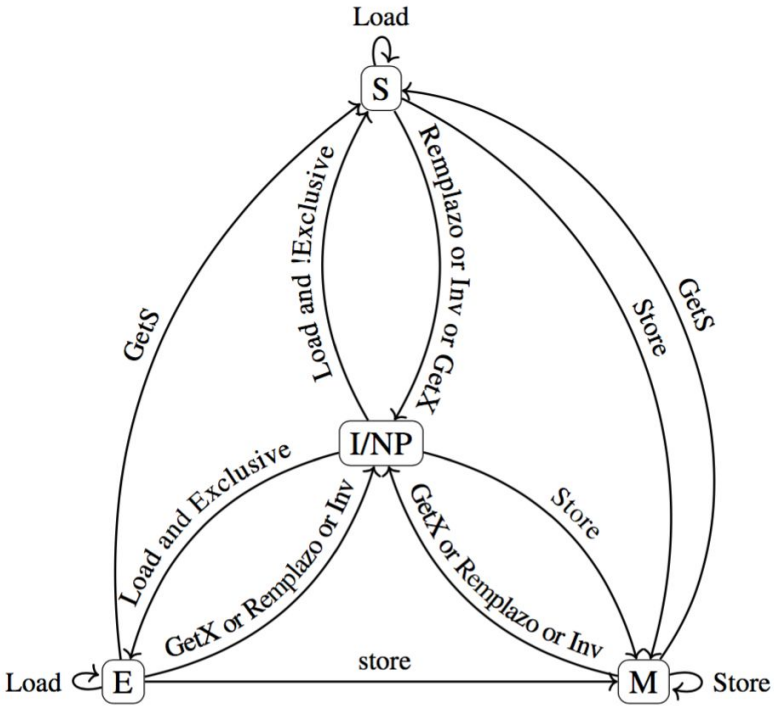



*Figura 1: Protocolo de coherencia MESI correspondiente a L1.*

Un bloque "*x*" en una cache L1 "*y*" puede encontrarse en los siguientes estados y sufrir las siguientes transiciones:

- Estados:
  - I/NP: El bloque "*x*" se encuentra invalidado o no está presente en la L1 "*y*".
  - S: El bloque "*x*" se encuentra en la L1 "*y*" y posiblemente en alguna otra L1, pudiendo ser leído.
  - M: La L1 *"y"* es la única propietaria del bloque *"x"*, pudiendo escribir y/o leer el bloque. El bloque *"x"* es potencialmente la única copia válida del sistema.
  - E: La L1 *"y"* es la única propietaria del bloque *"x"*, pudiendo escribir y/o leer el bloque *"x"*, pero el bloque *"x"* no ha sido escrito anteriormente.

- Transiciones:
  - Load and Exclusive: Petición de lectura del bloque *"x"* por parte del procesador, el bloque *"x"* no se encuentra en otra cache L1.
  - Load and !Exclusive: Petición de lectura del bloque *"x"* por parte del procesador, el bloque *"x"* se encuentra compartido por otra L1.
  - Load: Petición de lectura del bloque *"x"*.
  - Store: Petición de escritura por parte del procesador.
  - GetS: Petición de lectura de un bloque por parte de otra L1.
  - GetX: Petición de escritura de un bloque por parte de otra L1.
  - Reemplazo: Expulsión de un bloque de L1.
  - Inv: Invalidación de un bloque de L1.

**Nivel L2**

El segundo nivel de cache (L2 o LLC) es compartido entre todos los núcleos del procesador, guarda la información de inclusión - que bloques están compartidos y con qué cache L1 - y contiene todos los bloques existentes en L1. En la fig. 2 se puede observar los estados estables y las transiciones posibles de su protocolo. Una descripción completa del protocolo puede encontrarse en el anexo 1.2.

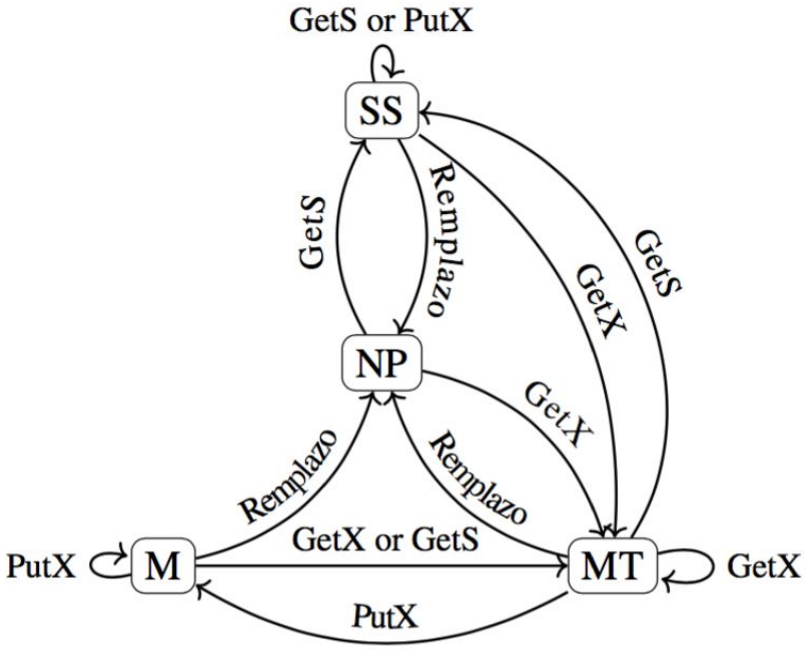



*Figura 2: Protocolo de coherencia MESI correspondiente a L2.*

A continuación se explica los estado estables en los que puede encontrarse un bloque *“x”* en la cache L2 y sus posibles transiciones:

- Estados:
  - NP: el bloque *“x”* no se encuentra presente en L2, por tanto tampoco está dentro de la jerarquía on-chip de memoria.
  - SS: el bloque *“x”* puede encontrarse en una o más caches L1 en modo sólo lectura. También es posible que el bloque *“x”* no se encuentre presente en ninguna L1.
  - M: el bloque *“x”* no se encuentra presente en ninguna L1.
  - MT: el bloque *“x”* se encuentra presente en una única cache L1 y L2 puede no tener una versión actualizada del mismo. Cualquier petición de lectura o escritura será redirigida a la L1 propietara del bloque y atendida por ella.

- Transiciones:
  - GetS: Petición del bloque *“x”* a L2 para lectura.
  - GetX: Petición del bloque *“x”* a L2 para escritura.
  - PutX: Expulsión del bloque *“x”* sucio de L1 para su actualización en L2.
  - Reemplazo: invalidación de todas las caches L1 propietarias del bloque *“x”* y expulsión del bloque *“x”* de L2.

## 3.2 Algoritmo de reemplazo Not Recently Reused (*NRR*)

Tradicionalmente, en todos los niveles de la jerarquía de memoria cache se han utilizado algoritmos que utilizan la localidad temporal para la selección del bloque víctima a reemplazar, un ejemplo de estos algoritmos es LRU. La base de estos algoritmos es que si un bloque se ha accedido recientemente, ese mismo bloque volverá a ser utilizado en un futuro cercano. Este tipo de algoritmos dan buenos resultados en los primeros niveles de la memoria cache debido a que la traza completa del procesador es observada. Sin embargo, tienen una eficacia bastante menor como política de reemplazo para la LLC.

La utilización de la localidad temporal en L1 provoca, con frecuencia, que todos los accesos a un bloque puedan ser atendidos por ella. En estos casos, la LLC solo recibirá la primera petición, cuando el bloque no se encuentre en L1. Por lo tanto, la utilización de una política como LRU en la LLC es ineficiente, debido a que el bloque no será accedido en un futuro cercano y su probabilidad de ser reemplazado aumentará. Esta ineficiencia, puede verse aumentada cuando se utilizan multiprocesadores cuyas cargas de trabajo interfieren unas con otras. Por ejemplo, un programa que realiza muchas peticiones seguidas a la LLC, aumenta la probabilidad de reemplazo de bloques más antiguos de otros programas, los cuales pueden ser reutilizados en un futuro[15]. La utilización de políticas basadas en reúso han demostrado una mejor eficiencia en la LLC evitando que se produzcan problemas como los anteriormente nombrados.

La idea principal de los algoritmos basados en reúso es que si un bloque ha sido usado por lo menos dos veces, este bloque es muy probable que vuelva a ser usado en un futuro cercano. Además, indica que los bloques más recientemente reusados son más útiles que los que fueron reusados con anterioridad [1].

Por otra parte, en las jerarquías inclusivas, los bloques se almacenan tanto en las caches privadas como en la cache compartida. Por ello, un reemplazo en la LLC sobre un bloque compartido provoca la invalidación en las caches privadas, y si el bloque está siendo usado por el procesador, un decremento del rendimiento.

La solución que se propone es utilizar el algoritmo *Not Recently Reused* (NRR)[5] que tiene en cuenta las observaciones puestas de manifiesto en el párrafo anterior. Esta política almacena en el estado de los bloques de LLC información de su reúso y presencia en los niveles privados. La inclusión de esta información requiere la utilización de un bit extra para indicar si el bloque presenta reúso.

Un bloque que se encuentra en LLC puede encontrarse en uno de los siguientes estados según su presencia y reúso:

- No reusado y Cacheado (NR---C): el bloque no presenta reúso (sólo ha sido accedido una vez) y se encuentra en una o más caches privadas.
- No Reusado y No cacheado (NR---NC): el bloque no presenta reúso y no se encuentra en ninguna cache privada.
- Reusado y Cacheado (R---C): el bloque presenta reuso (ha sido accedido dos o más veces) y se encuentra en una o varias caches privadas.
- Reusado y No Cacheado (R---NC): el bloque presenta reuso y no se encuentra en ninguna cache privada.

El comportamiento de la política se puede observar en el siguiente diagrama:

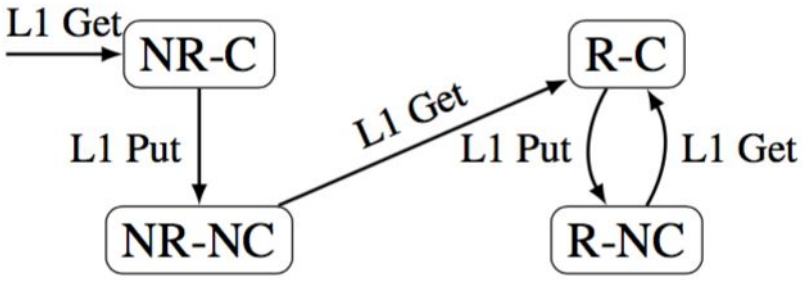



*Figura 3: Diagrama de estados NRR según reuso e inclusión.*

Este algoritmo asigna la máxima protección, mínima probabilidad de ser reemplazado, a aquellos bloques presentes en las caches privadas y entre los no presentes, a aquellos que presentan reúso. Por lo tanto, la probabilidad de reemplazo de un bloque según su reuso e inclusión es la siguiente (de mayor probabilidad a menor), ver fig. 4: No Reusado y No Cacheado (NR-NC) > Reusado y No Cacheado (R-NC) > No Reusado y Cacheado (NR-C) > Reusado y Cacheado (R-C).

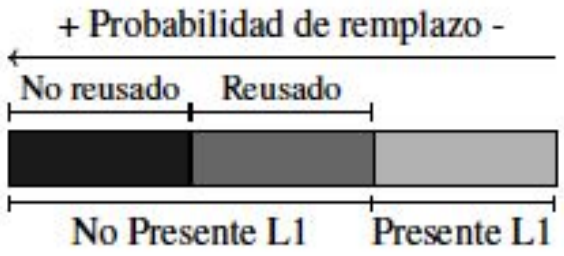



*Figura 4: Probabilidad de un bloque de ser remplazado con NRR*

La implementación del algoritmo NRR para el segundo nivel de cache se corresponde con la siguiente descripción en alto nivel [5]:

- **Acierto** en cache sobre un bloque *b*:
  1. Poner bit de reuso del bloque *b* a ’1’.
  2. Poner el bit de presencia del bloque *b* a ’1’.
  3. Si todos los bits de reuso de las demás entradas del mismo set con el bit de presencia a ‘0’ están a ‘1’, se ponen a ’0’, exceptuando el del bloque *b.*

- **Fallo** en cache sobre un bloque *b*:
  1. Se busca aleatoriamente un bloque víctima con el bit de presencia a ’0’ y con el bit de reuso a ’0’.
  2. Si se ha encontrado un bloque saltar a 4.
  3. Se busca aleatoriamente un bloque víctima con el bit de reuso a ’0’.
  4. Reemplazo del bloque seleccionado.
  5. Se inserta el bloque *b*.
  6. El bit de presencia del bloque *b* se pone a ’1’.
  7. El bit de reuso del bloque *b* se pone a ’0’.

### 3.2.1 Reemplazo no silencioso

En la versión por defecto del protocolo de coherencia (MESI) del simulador Gem5, las caches privadas no avisan a L2 (cache compartida, LLC) cuando efectúan un reemplazo de un bloque limpio; por lo que la LLC no posee información precisa sobre la presencia de un bloque en los niveles inferiores.

NRR guarda información de los bloques sobre su reúso y presencia en las caches privadas. Para que las prestaciones de esta política sean óptimas, es necesario que la LLC tenga accesible dicha información de manera precisa en todo momento, por ello, es necesario que las caches privadas notifiquen a la LLC cuando expulsan un bloque limpio.

Para conseguir que la *LLC* tenga información precisa se ha modificado el protocolo de coherencia de manera que siempre que se produzca un reemplazo en una cache privada L1, se envíe un mensaje *PutX* a la LLC. De esta manera la cache compartida actualiza el vector de bits de presencia cada vez que recibe este mensaje.

**Actualización de *Sharers* de un bloque**

Cada bloque que se encuentra en la LLC tiene asociado a él una estructura de datos (*sharers*) que le indica si está compartido, y en caso de estarlo, con que cache privada. Por tanto, cada vez que a LLC le llega una notificación de reemplazo (PutX) de un bloque en L1, la lista de *sharers* ha de actualizarse; eliminandose de la estructura la información que asocia el bloque con la L1 que ha sufrido el reemplazo.

Esta actualización de la lista de *sharers* en el protocolo original solo se realizaba, si el bloque reemplazado estaba sucio. Con la implementación del reemplazo no silencioso, también se realiza por el reemplazo en L1 de un bloque limpio. Si la L1 que reemplaza el bloque es la única propietaria del mismo, no está compartido por ninguna otra L1, el bloque cambiará al estado M en la LLC.

**Cruce de mensajes**

Con reemplazo no silencioso aparece un posible bloqueo de la jerarquía de cache, debido a un cruce de mensajes, ver fig. 6. El cruce de mensajes ocurre, por ejemplo, cuando en la cache privada y en la cache compartida, en paralelo, se produce un reemplazo en el mismo bloque. Como consecuencia de ello, ambas comienzan un proceso de invalidación sobre el bloque. La cache compartida envía invalidación de bloque a todas las caches privadas que poseían ese bloque y la cache privada le envia PutX a la cache compartida, quedando, ambas, en un estado intermedio esperando sendos ACKs.

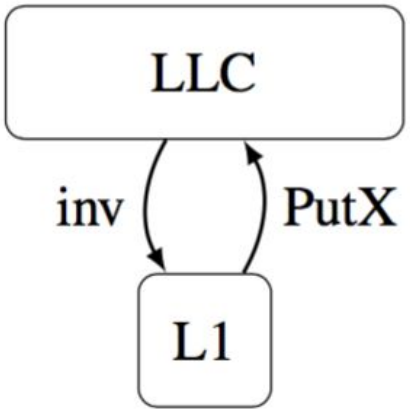



*Figura 6: Cruce de mensajes en remplazo no silencioso*

La solución adoptada, para evitar este bloqueo, consiste en modificar el estado transitorio de la cache privada en el que se queda esperando el ACK del *PutX*. En ese estado, si se recibe un invalidación, la cache privada se comportará como si se tratara de un ACK y se enviará el correspondiente ACK de la invalidación a la cache compartida, quedando ambas desbloqueadas.

## 3.3 Block Disabling

Para implementación de *BD*, previamente explicado en la sec. 2.2, se han realizado los siguientes cambios en el simulador:

1. Inclusión de un *array* auxiliar en el objeto que enumera las propiedades con las que cuenta cada entrada de la memoria cache.
2. En el momento de inicialización de las memorias caches se elige de manera probabilista si cada una de las entradas va a estar habilitada o deshabilitada, 75% de probabilidades para estar deshabilitada y 25% de posibilidades de estar habilitada. Esta probabilidad se corresponde con la que tienen las celdas C3[2] de fallar ante una bajada de tensión.
3. Modificación del algoritmo de reemplazo para que elimine las entradas deshabilitadas del conjunto de víctimas potenciales.

## 3.4 BDOT

*BDOT* permite, como ha sido explicado anteriormente, mantener la inclusión de directorio de forma que las entradas defectuosas puedan ser incluidas en las operaciones de búsqueda y reemplazo. De esta forma, un bloque que reside en una entrada defectuosa en la LLC (sólo etiqueta) pueda ser almacenado en las caches privadas.

El protocolo de coherencia también tiene que adaptarse de forma que sea capaz de discernir aquellas entradas sanas — entradas *D* — y las defectuosas — entradas *T* — y modificar su comportamiento según las mismas. Para diferenciar las entradas solo hace falta la inclusión de un bit en la información asociada a la entrada, de manera equivalente a como se realiza en *BD*.

Por tanto un conjunto en BDOT con asociatividad *N* contendrá entradas *T* que solo almacenarán etiquetas y entradas *D* que almacenarán etiquetas y datos. La suma de ambos, *T* + *D* = *N*, dará como resultado el número de entradas del conjunto.

Cuando llegue una petición sobre un bloque *D* ésta será tratada de manera estándar, en cambio, una petición de lectura o escritura sobre un bloque *T* será reenviada a las caches privadas si el bloque se encuentra en estado SS o MT; o será reenviada a memoria principal si se encuentra en estado M. Si la petición es un reemplazo de un bloque limpio de L1 se actualizará la información de presencia asociada al mismo en L2. Si, por el contrario, es un bloque sucio, éste será enviado a memoria principal. Un diagrama de flujo de estas acciones puede observarse en la fig. 7.

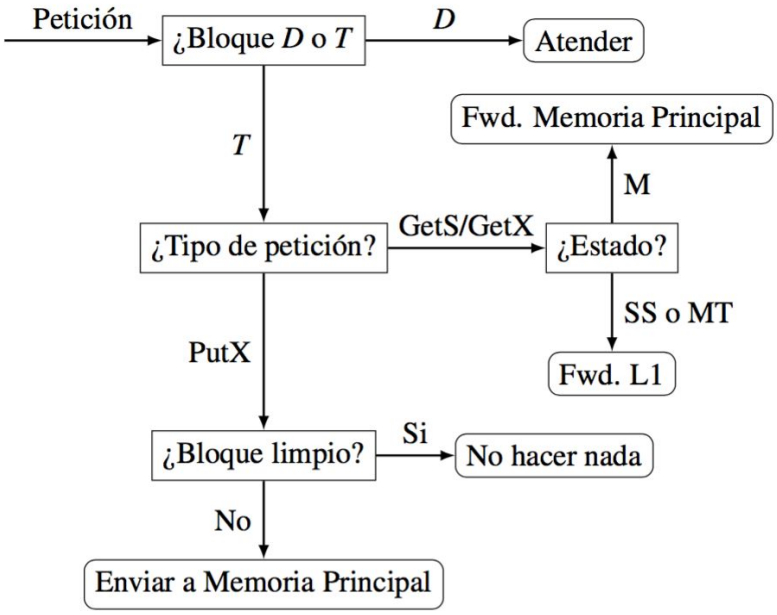



*Figura 7: Diagrama de flujo de las peticiones sobre bloques T o D.*

Por lo tanto el nuevo diagrama de estados estables del protocolo de coherencia para el segundo nivel de memoria cache se puede observar en la fig. 8, al visto originalmente en la sección 3.1.1 hay que añadirle las nuevas transiciones posibles:

- GetXT: petición de escritura sobre un bloque T.
- GetST: petición de lectura sobre un bloque T.
- PutXT: expulsión de un bloque T.

Además para mejorar el entendimiento del diagrama se han añadido a cada transición una serie de acciones asociadas:

- Fwd. L1: enviar petición a L1 de forma que esta envíe su copia a la L1 solicitante.
- Fwd. Mem: petición a memoria principal del bloque solicitado.
- Send.: L2 envía el bloque a la L1 solicitante.
- Send. Mem: envio del bloque a memoria principal.
- Copy: actualización del bloque de L2 con el llegado de L1.

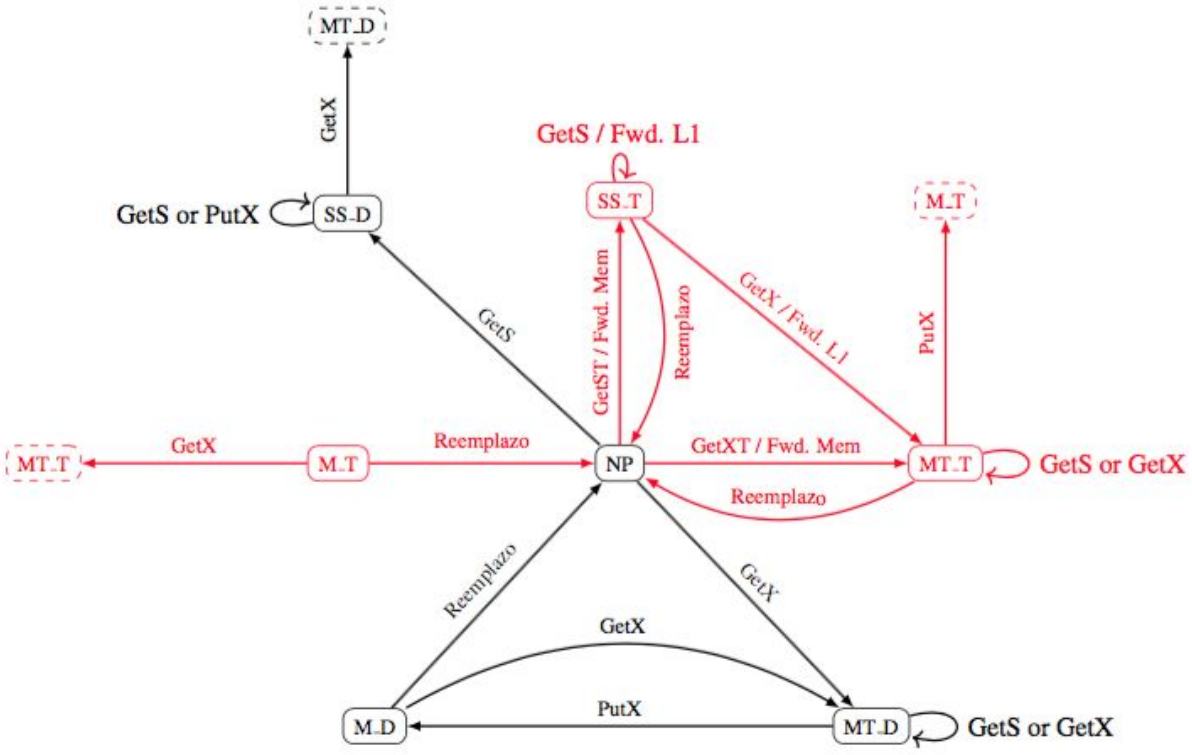



*Figura 8: Protocolo de coherencia de L2 con BDOT*

Respecto al protocolo de coherencia de las caches L1 hay que realizar una mínima modificación de forma que sea capaz de atender peticiones GetS desde el estado SS - bloque compartido entre una o más memorias caches -.

## 3.5 BDOT: reemplazo específico

Como se ha puesto de manifiesto en la sección 2.5, la principal limitación del esquema BDOT es que no tiene en cuenta el tipo de entrada en la asignación de los bloques a ellas. Si un bloque con un patrón de reuso ha sido asignado a una entrada defectuosa (T), todas las peticiones sobre él serán redirigidas a memoria. Por lo tanto, es necesario implementar BDOT junto con una política de gestión de contenidos capaz de distinguir estas situaciones.

Uno de los aspectos claves para mejorar el rendimiento de un subsistema de memoria con vías deshabilitadas, es el intercambio de los bloques más utilizados entre entradas *T* y *D*. Para ello necesitamos tener un algoritmo de selección de víctima y un protocolo que permita el intercambio de bloques entre entradas de la misma cache. El objetivo de esta nueva política es asegurar que un bloque que tiene un número elevado de peticiones en la cache L2 resida en una entrada de tipo D y, por lo tanto, no se sufra la penalización de ir a memoria principal para buscar el bloque.

### 3.5.1 Algoritmo de selección de víctima Swap

El algoritmo de selección de víctima para el intercambio es una modificación del algoritmo de reemplazo presentado anteriormente (NRR). Su objetivo es garantizar que las entradas no defectuosas (D) son ocupadas por aquellos bloques que no están presentes en caches privadas y que muestran un patrón claro de reuso, por tanto, que sea más probable que sean usados en un futuro cercano.

Para ello, es necesario desdoblar en dos cada estado de la política de reemplazo base: uno para entradas de etiquetas, *T*, y otro para entradas de etiqueta y datos, *D*, tal y como se muestra en la fig. 9. Mientras un bloque recorre los estados NR-C, NR-NC y R-C puede estar alojado en una entrada *T* o *D*, en función de la vía que inicialmente le asigna el algoritmo de reemplazo. Sin embargo para proteger a los bloques con reuso, cuando un bloque en estado R-C es expulsado de la cache privada le aseguramos una entrada de tipo *D* (estado R-NC-D), debido a las alta probabilidad existente de que el bloque vuelva a ser utilizado.

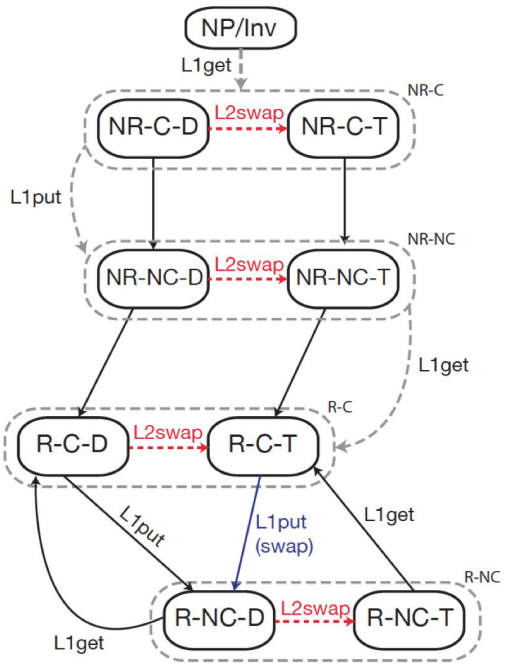



*Figura 9. Estados según reúso en inclusión de un bloque en LLC con BDOT.*

Esta transición normalmente requiere que otro bloque sea degradado de una entrada *D* a la entrada *T* que queda libre, lo que implica la perdida de los datos del bloque de la LLC. Este intercambio de bloques requiere que el algoritmo de reemplazo sea capaz de seleccionar un bloque en estado *D* y lo degrade a estado *T*. El objetivo final de este algoritmo es maximizar los contenidos dentro del chip, degradando prioritariamente aquellos bloques presentes en caches privadas.

Con el algoritmo de reemplazo específico un bloque puede encontrarse en los siguientes estados: No reusado, No cacheado y en entrada *D* (NR-NC-D), No reusado, No cacheado y en entrada *T* (NR-NC-T), No reusado, cacheado y en entrada *D* (NR-C-D), No reusado, cacheado y en entrada *T* (NR-C-T), Reusado, cacheado y entrada *D* (R-C-D), Reusado, cacheado y en entrada *T* (R-C-T), Reusado, no cacheado y en entrada *D* (R-NC-D) y Reusado, no cacheado y en entrada *T* (R-NC-T).

Para conseguir los objetivos nombrados anteriormente, se establece el siguiente orden de prioridad de que un bloque sea degradado: NR-C-D > R-C-D > NR-NC-D > R-NC-D. Como puede observarse, es imposible que un bloque en estado T sea degradado, pues ya ocupa por sí mismo una entrada sin prioridad.

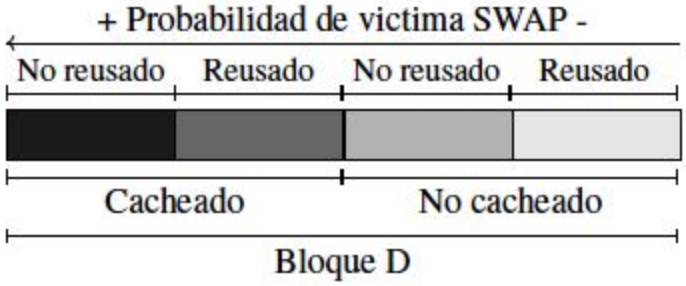



*Figura 10. Probabilidad de un bloque de ser degradado en un intercambio.*

### 3.5.2 Modificación del protocolo de coherencia

Para poder realizar intercambio de bloques entre entradas de una memoria cache es necesario que el protocolo de coherencia soporte y permita el intercambio de mensajes donde el emisor y el receptor es la misma memoria cache, además de la inclusión de nuevos estados y transiciones que permitan este intercambio.

Las acciones para realizar un intercambio (*Swap*) de un bloque *h* que reside en una entrada de tipo T son las siguientes:

1. Cuando un bloque es expulsado de una cache privada (L1_PutX) se comprueba si es un candidato válido, es decir, muestra reuso, no está compartido por ninguna cache privada más y está en una entrada *T*.
2. Si es un candidato válido, se inicia el intercambio copiando el bloque procedente de la cache L1 en la nueva entrada D y enviando a memoria el bloque que residía en dicha entrada.
3. Si no lo es, se envía el bloque procedente de la cache L1 a memoria atendiendo de la manera oportuna un PUTX sobre un bloque T.

La implementación de este protocolo no supone ningún cambio en los diagramas de estados estables del protocolo de coherencia anteriormente vistos. Sin embargo, sí que requiere modificaciones y creación de numerosos estados intermedios. En el anexo 2. se explican con detalle todas las modificaciones que ha sufrido el protocolo.

## 3.6 Desacoplo de etiqueta y datos

Como se ha puesto de manifiesto anteriormente, ver sección 2.6, el desacoplamiento [8] de etiquetas y datos consiste en desvincular la relación entre las etiquetas y los datos permitiendo que cualquier etiqueta apunte a cualquier contenedor de datos. Esta modificación requiere un sistema de punteros que relacione cada etiqueta con el contenedor de datos correspondiente.

Por lo que respecta al algoritmo de selección de víctima para un intercambio, necesita ser modificado de forma que la búsqueda de la víctima no esté limitada solamente a las vías del conjunto al que pertenece el bloque. El rango de la búsqueda queda ampliado a la totalidad de las entradas tipo D de toda la cache L2.

# 4. Metodología

En esta sección presentamos la metodología utilizada, incluyendo la comprobación cruzada de código, características del sistema modelado, entorno de simulación, modelos simulados y métricas utilizadas para cuantificar los beneficios.

## 4.1 Comprobación cruzada de código

Uno de los principales problemas de la investigación y de la implementación de nuevos modelos en simuladores es el desconocimiento final a cerca de la corrección de lo modelado.

Para comprobar que la implementación es correcto y se ajusta al modelo conceptual descrito anteriormente se ha llevado a cabo un análisis cruzado del código. Que consiste en la revisión del código implementado por el otro autor utilizando técnicas como *hints* (comprobaciones sobre el modelo en tiempo de ejecución con el objetivo de que no se produzcan eventos y situaciones incoherentes con el modelo conceptual) o el análisis de estadísticas y resultados tras la ejecución de las aplicaciones.

Algunos ejemplos son:

- Volcado de la información asociada a los bloques existentes en L1 y en L2 sobre un fichero cada diez mil ciclos. Posteriormente, análisis del fichero para comprobar la coherencia de la información de los bloques, es decir, que cada bloque de L1 se encuentra en L2 y L2 tiene la información de seguimiento del bloque correcta. De esta forma comprobamos la correcta implementación del reemplazo no silencioso.
- Análisis en profundidad del código y de los diagramas de estado del protocolo de coherencia generados por el compilador.

- Problema encontrado: Cuando la cache L1 hacía un reemplazo de un bloque sucio que correspondía con una entrada defectuosa (T), el bloque tenía que ser enviado a memoria. En esa transición, la LLC enviaba el ACK a la L1. Si se envía ese ACK, la cache L1 sale del estado intermedio quedando cualquier petición de otro core sobre ese bloque insatisfecha ya que el bloque se encuentra en camino entre la LLC y memoria.
- Solución propuesta: La cache LLC no envía el ACK a la cache L1 hasta que la memoria no haya recibido el bloque y enviado el correspondiente ACK a la LLC. Así, la cache L1 puede satisfacer cualquier petición sobre ese bloque.

- Comprobación en tiempo de ejecución que no se satisface directamente desde la LLC una petición a un bloque *T* en el modelo BDOT.
  - *Hint*: Si durante la ejecución de una aplicación, la memoria cache L2 satisface directamente una petición de un bloque albergado en una entrada de tipo T en vez de hacer forward a memoria o a una cache L1 salta un error que nos indica el fallo.

- Comprobación en tiempo de ejecución que un bloque *T*, en el modelo BDOT con reemplazo específico, no puede ser víctima de un reemplazo, de la misma forma, un bloque *D* no puede iniciar un intercambio.
  - *Hint*: Cuando se procede al intercambio entre dos bloques en el protocolo, si la víctima del intercambio es un bloque albergado en una entrada de tipo T, salta un error que nos indica el fallo.
  - *Hint*: Cuando se procede al intercambio entre dos bloques en el protocolo, si el bloque que ha iniciado el intercambio es un bloque albergado en una entrada de tipo D, salta un error que nos indica el fallo.

- Utilización de las estadísticas para comprobar la coherencia en los resultados devueltos. A continuación se presentan unos ejemplos de cómo pueden ser analizadas las estadísticas para concluir que los modelos implementados son correctos:
  - El número de bloques traídos a L1 coincide con el número de notificaciones de reemplazo en L1 a L2.
  - El número de invalidaciones de L2 a L1 es menor con reemplazo silencioso que sin él.
  - El número de accesos a memoria principal con LRU es el mismo cuando se utiliza reemplazo silencioso y no silencioso.
  - El número de accesos a memoria principal cuando se utiliza *block disabling* es mayor que cuando no se utiliza.

## 4.2 Sistema modelado

Nuestro sistema de referencia es un chip multiprocesador (CMP) teselado con una jerarquía cache inclusiva de dos niveles. Cada tesela contiene un núcleo y un primer nivel de cache (L1) dividida en instrucciones y datos. El segundo nivel (L2 o LLC) es compartido y está distribuido entre los cuatro núcleos. Las teselas están interconectadas por medio de un *crossbar*. La tabla 1, recoge los principales parámetros de configuración del sistema de referencia.

| | |
|---|---|
| Núcleos | 4, Alpha 21264, 2GHz, 4 instrucciones por ciclo |
| Protocolo de Coherencia | MESI, directorio |
| L1 cache | Privada, 64KB de datos y 32 KB de instrucciones, asociatividad 2, LRU, 2 ciclos de acceso en caso de acierto |
| L2 cache | Compartida, inclusiva, 1MB por núcleo, asociatividad 16, 15 ciclos de acceso en caso de acierto |
| Memoria | 1GB por núcleo, un canal |
| Red on-chip | Cossbar |

*Tabla 1. Configuración del sistema de referencia.*

El protocolo de coherencia se implementa sobre un directorio con estados MESI como se ha nombrado en los apartados anteriores. La caches L1 están construidas con celdas robustas y funcionan a voltajes bajos sin errores.

## 4.3 Modelos simulados

Los modelos simulados han sido los siguientes:

- **BASE:** Sistema base y LRU.
- **B_NRR:** Sistema base, reemplazo no silencioso y NRR .
- **BD:** Sistema base con *Block Disabling*, reemplazo no silencioso y NRR.
- **BDOT:** BDOT, reemplazo no silencioso y NRR.
- **BDOT_RE:** BDOT, reemplazo no silencioso y NRR específico.
- **BDOT_CD:** BDOT, reemplazo no silencioso, NRR específico y LLC desacoplada.

## 4.4 Entorno de simulación y cargas de trabajo

Con respecto al entorno de simulación, usamos Gem5 con Ruby para modelar la jerarquía de memoria on-chip y la red de interconexión, y *System Call Emulation (SE)* para ejecutar aplicaciones sin sistema operativo.

Para la simulación monocore utilizamos un conjunto de cuatro benchmarks de SPEC CPU 2006, utilizando aquellos que presentan una utilización intensa de la jerarquía de memoria cache. En concreto se han utilizado los siguientes: dealII, omnetpp, soplex y milc. Para asegurarnos de que ninguna de las cargas se encuentra en fase de inicialización en la simulación monocore, cada programa ha sido analizado mediante *simpoints*, y se crearon *checkpoints* en sus zonas más representativas. A partir de ahí, ejecutamos simulaciones que incluyen 100 millones de instrucciones para calentar la jerarquía de memoria y 200 millones de instrucciones de simulación detallada.

Para la simulación multicore, se ha confeccionado una mezcla con los cuatro benchmarks utilizados para la simulación monocore. Con el objetivo de ejecutar en detalle las partes más representativas de las aplicaciones, se han saltado mil millones de instrucciones de la mezcla calentado la memoria cache y se han ejecutado 200 millones de instrucciones de manera detallada.

Calculamos si una entrada de la cache L2 falla o no de forma aleatoria e independiente del resto de entradas, cada una de las entradas tiene una probabilidad del 75% de ser defectuosa. Para poder comparar con el *block disabling* se asume que al menos existe una vía no defectuosa en cada conjunto.

## 4.5 Métricas

Por último, para comparar los distintos modelos implementados, vamos a utilizar dos métricas de rendimiento: fallos de la LLC por cada mil instrucciones (MPKI) e instrucciones por ciclo (IPC).

El MPKI nos muestra el aumento del tráfico de memoria fuera del chip. Para calcular el MPKI multiplicamos el número total de peticiones a memoria principal por mil y dividimos el resultado por el número de instrucciones ejecutadas.

# 5 Resultados

En esta sección presentamos los resultados de la evaluación de nuestra técnica de gestión de contenido utilizando las métricas expuestas en la sección 4.5. Nuestro sistema de referencia es una LLC ideal, implementada con celdas robustas, capaces de operar sin fallos a muy bajo voltaje y sin ningún tipo de penalización.

La tabla 2 muestra los resultados para una simulación monocore de las cuatro cargas de trabajo nombradas en las secciones anteriores. En ella se puede apreciar para cada aplicación el MPKI y el IPC de cada uno de los modelos.

La tabla 3 muestra los resultados para una simulación multiprocesador con una mezcla de los cuatro benchmarks. En ella se presenta para cada aplicación de la mezcla y para cada modelo implementado la métrica IPC y la métrica MPKI de la mezcla entera.

Los mismos datos se presentan en las figuras 11, 12, 13 y 14. En estas gráficas se representan valores relativos al sistema base.

La diferencia esperada entre el sistema base y el NRR no es mucha. Esto es debido a que el NRR funciona mejor con las aplicaciones que muestran reuso pero el LRU es capaz de establecer un orden completo con respecto a la localidad temporal de los bloques. En los resultados monocore, puede apreciarse esa diferencia en MPKI que llega a un máximo en *soplex* con un factor de 1,35. La métrica de IPC se mantiene casi invariante. En los resultados multicore puede apreciarse también esa diferencia en MPKI con un factor 1,02. La métrica de IPC se mantiene invariante en todas las aplicaciones con pequeñas variaciones.

La diferencia esperada entre los dos primeros modelos (Base y NRR) y BD sí que es notable, ya que con *block disabling* se pierde el 75% de la capacidad de la cache y, lo que es peor, se pierde asociatividad. Esto implica que no se pueda mantener el estado de coherencia de los bloques en la LLC y, en consecuencia, se produzcan numerosas invalidaciones en L1. En los resultados monocore, puede apreciarse esta diferencia tanto en MPKI como en IPC con un factor máximo en MPKI respecto a Base de 23,58 en *soplex* y un factor mínimo en *dealII* de 2,77. En los resultados multicore, el factor en MPKI es de 9,58 y en IPC de 1,58 en *dealII*.

Con BDOT se soluciona el problema de la pérdida de asociatividad en la cache LLC ya que se puede guardar el estado de coherencia de los bloques en la LLC lo cual debería disminuir el número de invalidaciones. Esta mejora frente a BD debería verse reflejado en los resultados. En los resultados monocore puede apreciarse mejora en MPKI e IPC. Los factores máximos respecto a BD de MPKI e IPC se alcanzan en *soplex* con 1,71 y 1,62 respectivamente. En los resultados multicore el factor MPKI es de 2,57 para la mezcla y el factor IPC máximo se encuentra en *dealII* con 1,32.

En el esquema BDOT puede ocurrir que a un bloque que presente patrón de reuso se le haya asignado una entrada tipo T. Por consiguiente, todas las peticiones sobre él tendrán que ser satisfechas desde memoria. La política de reemplazo específico es capaz de distinguir situaciones como la descrita anteriormente, consiguiendo una mejor gestión de los contenidos dentro de la LLC, consiguiendo una disminución de los accesos a memoria. En los resultados monocore puede apreciarse esta mejora tanto en MPKI como en IPC cuyos factores máximos (entre BDOT_RE y BDOT) se alcanzan en *omnetpp,* 1,75 y *dealII*, 1,29 respectivamente. En los resultados multicore, el factor MPKI es de 1,38 y el IPC máximo se presenta en *dealII* con 1,15.

Por último, la cache desacoplada permite optimizar las políticas de gestión de contenidos de la LLC buscando una víctima potencial de un intercambio en toda la cache. La búsqueda de esta víctima en el esquema no desacoplado estaba limitado al conjunto al que pertenece. Esta mejoría puede apreciarse en los resultados monocore tanto en MPKI como en IPC cuyos factores máximos (entre BDOT_CD y BDOT_RE) se alcanzan en omnetpp, 1,32 y 1,41 respectivamente. En los resultados multicore, el factor MPKI es de 1,96, sin embargo, el IPC queda prácticamente invariante.

| | **soplex** | | **omnetpp** | | **milc** | | **dealII** | |
|---|---|---|---|---|---|---|---|---|
| | MPKI | IPC | MPKI | IPC | MPKI | IPC | MPKI | IPC |
| **Base** | 0,31 | 0,73 | 0,43 | 1,12 | 5,67 | 0,41 | 0,36 | 1,66 |
| **B_NRR** | 0,42 | 0,73 | 0,44 | 1,11 | 5,67 | 0,41 | 0,31 | 1,65 |
| **BD** | 7,13 | 0,34 | 2,40 | 0,95 | 10,46 | 0,17 | 3,23 | 0,60 |
| **BDOT** | 4,18 | 0,55 | 1,54 | 0,87 | 8,14 | 0,23 | 2,32 | 0,85 |
| **BDOT_RE** | 3,56 | 0,67 | 0,88 | 0,76 | 7,35 | 0,27 | 1,75 | 1,10 |
| **BDOT_DC** | 3,34 | 0,64 | 0,66 | 1,07 | 6,12 | 0,36 | 1,35 | 1,39 |

*Tabla 2. Resultados simulación monocore.*

| **Mezcla** |
|---|

| | | **dealII** | **omnetpp** | **soplex** | **milc** |
|---|---|---|---|---|---|
| | MPKI | IPC (0) | IPC (1) | IPC (2) | IPC (3) |
| **LRU** | 0,38 | 1,69 | 1,16 | 0,88 | 0,13 |
| **NRR** | 0,39 | 1.69 | 1.15 | 0,88 | 0,13 |
| **BD** | 3,64 | 1,07 | 0,84 | 0,62 | 0,13 |
| **BDOT** | 1,41 | 1,42 | 1,02 | 0,75 | 0,13 |
| **BDOT_RE** | 1,02 | 1,64 | 1,14 | 0,85 | 0,13 |
| **BDOT_CD** | 0,52 | 1,66 | 1,15 | 0,86 | 0,13 |

*Tabla 3. Resultados simulación multicore.*

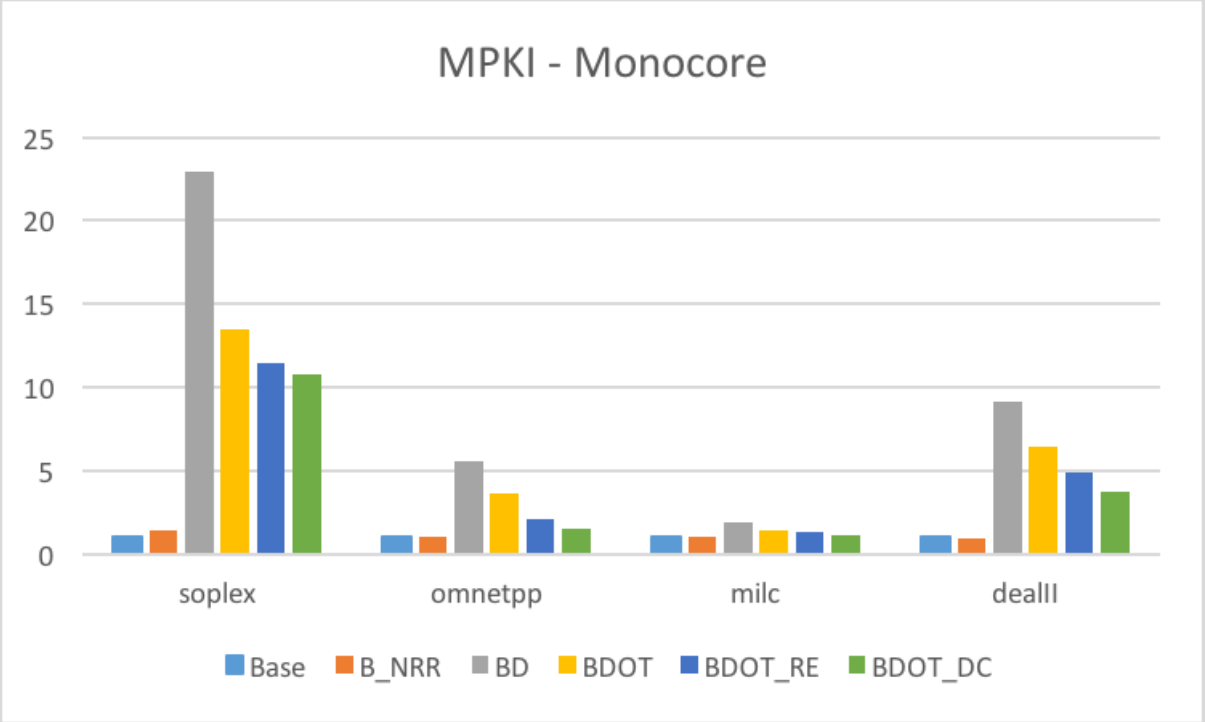



*Figura 11. MPKI normalizado para las cargas de trabajo monocore*

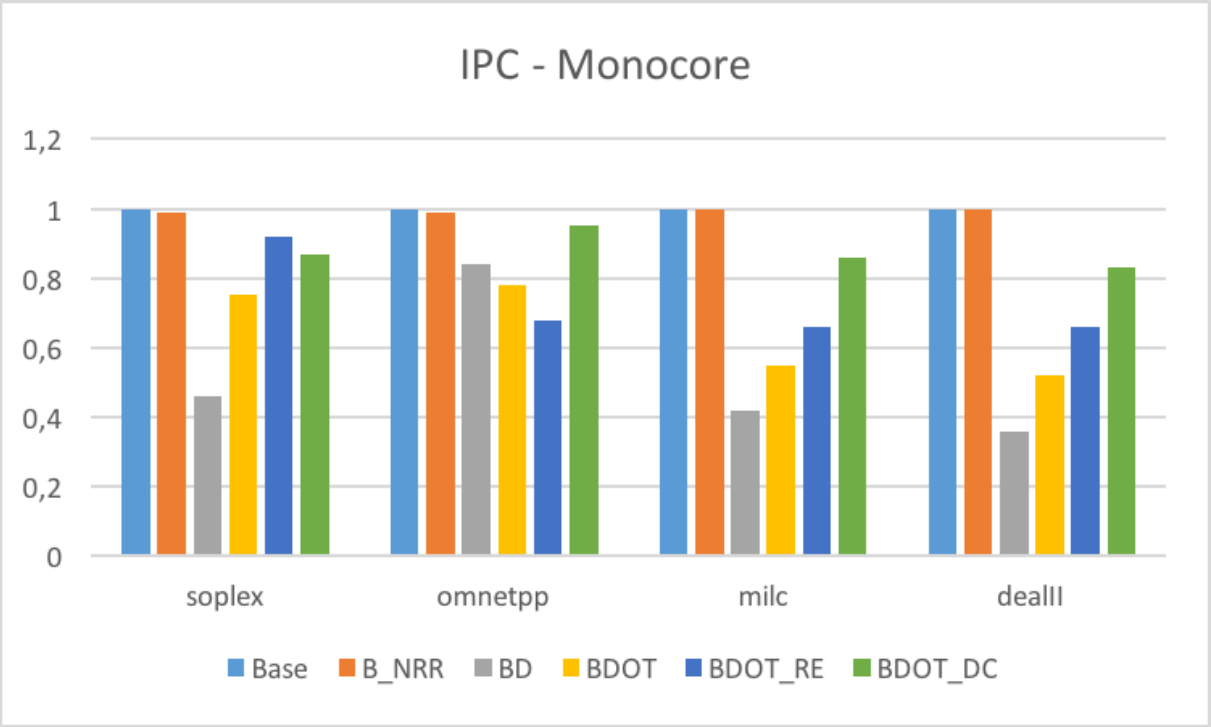



*Figura 12. IPC normalizado para las cargas de trabajo monocore*

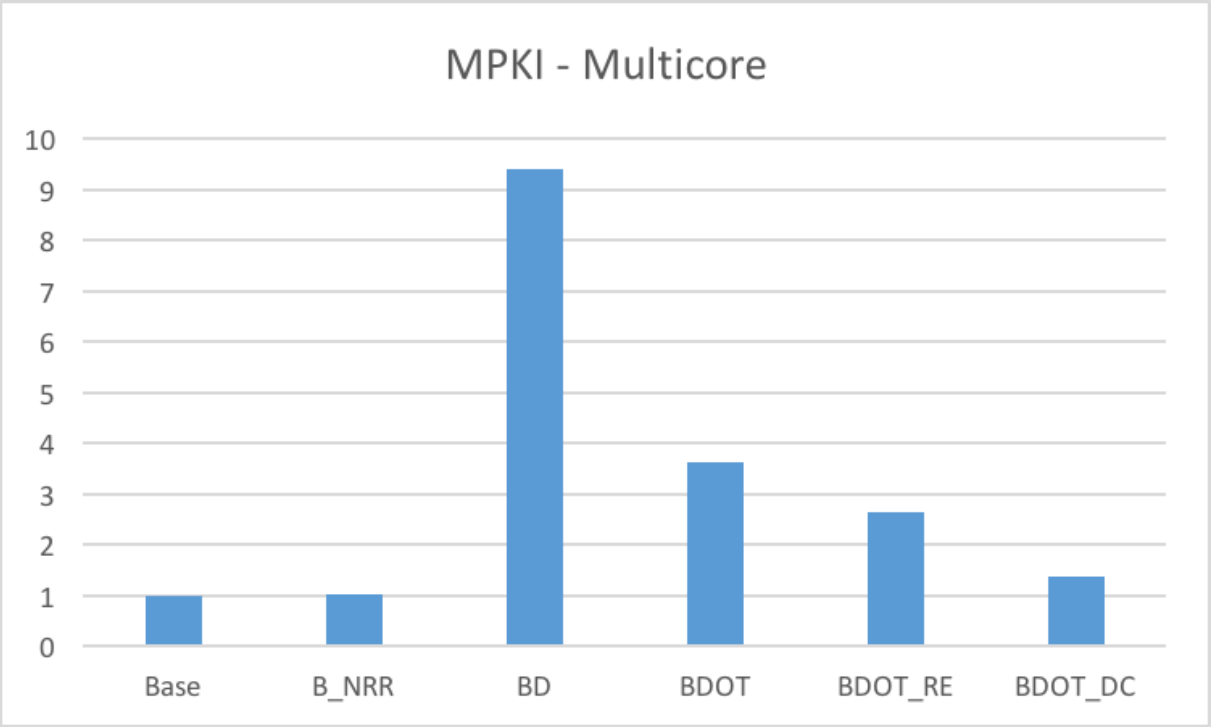



*Figura 13. MPKI normalizado para la carga de trabajo multicore*

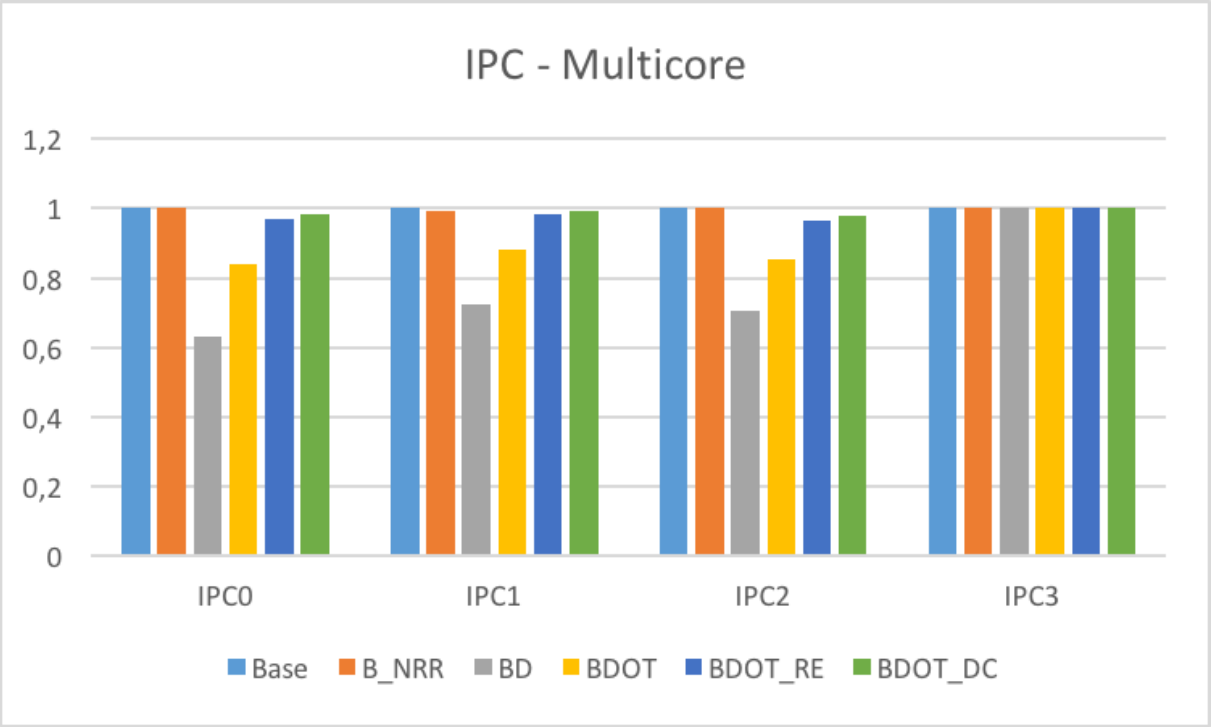



*Figura 14. IPC normalizado para la carga de trabajo monocore*

# 6 Conclusiones

A continuación se explican los principales problemas encontrados durante el desarrollo del trabajo y las conclusiones a nivel técnico y personal de la realización del mismo.

## 6.1 Problemas encontrados

Durante el desarrollo del proyecto, nos hemos encontrado con distintos escollos, alguno de ellos se resolvieron con relativa facilidad, y otros han supuesto graves problemas para el desarrollo del mismo. Alguno de los problemas más significativos encontrados han sido:

- La falta de documentación oficial del simulador, lo que ha complicado el aprendizaje del mismo. Esta falta de información técnica, ha hecho necesaria la utilización de listas de correo y el contacto con otros usuarios del simulador.
- La ausencia de *simpoints* y *checkpoints* para procesadores multicores. Lo que ha obligado a la utilización de técnicas alternativas para poder simular en procesadores multicores.
- La incapacidad de lanzar simulaciones en *atps*. Esto fue debido a que el *front-end* del cluster albergaba un sistema diferente al del resto de los nodos, por lo tanto,

los binarios generados por el simulador en el nodo principal no eran compatibles en el resto de nodos. Además, estos nodos no cuentan con las dependencias mínimas para la compilación y ejecución de Gem5. A su vez, para la instalación de estas dependencias era necesaria la actualización del sistema operativo. La ausencia de los nodos de atps ha tenido un impacto directo en el transcurso del proyecto debido a:

- El intento infructuoso de instalación de dependencias del simulador durante dos semanas.
- La potencia de cálculo se vio limitada al procesador de cuatro cores del *front-end* del cluster para ejecutar. Así pues, la capacidad de simulación quedó acotada a un máximo de cuatro simulaciones simultáneas, lo que ha hecho inviable la simulación de todos los benchmarks.

## 6.2 Cumplimiento de objetivos

A continuación se resaltan los objetivos planteados para este trabajo y cumplidos de manera satisfactoria:

- El objetivo principal del proyecto era el desarrollo de una nueva propuesta de investigación para el funcionamiento de memorias caches a muy bajo voltaje. Este objetivo se ha cumplido tras la implementación de diversos modelos y su posterior modificación.

- Se ha aprendido a utilizar el simulador Gem5, conociendo los aspectos más importantes del mismo, como son: la estructura general del simulador, el subsistema de memoria y los diversos protocolos de coherencia, los parámetros de configuración de la memoria, los distintos parámetros para la simulación de cargas de trabajo, y el sistema de depuración del sistema mediante trazas.

- Otro objetivo de este trabajo era la utilización de *simpoints* y *checkpoints* para la realización de simulaciones monocore para el análisis de las cargas de trabajo y la simulación ciclo a ciclo de sus partes más representativas.

- Se ha aprendido a realizar *fast-forward* con el fin de ser capaces de saltar la fase de inicialización de los programas y únicamente simular de manera detallada las partes representativas de las mismas.

- Otro de los objetivos planteados en el proyecto era aprender a simular cargas de trabajo real sobre un simulador y analizar las estadísticas resultantes. Tras la simulación de una mezcla para el sistema multiprocesador y diversas cargas de trabajo para el sistema monocore, se puede dar por cumplido este objetivo. Sin embargo, no se ha podido podido hacer una simulación exhaustiva de todas las cargas de trabajo como se había planteado inicialmente.

Por otro lado, una serie de objetivos planteados al comienzo del proyecto no han podido cumplirse completamente, y por tanto pueden mejorarse en un futuro:

- La utilización de *condor* y *atps* para la simulación de las cargas de trabajo. Este objetivo no ha podido cumplirse por los motivos presentados en la sección anterior.
- La simulación completa de todos los spec disponibles. No disponer de la potencia de cálculo necesaria (cluster *atps*) necesario para ello, ni del tiempo necesario.
- Mejorar los modelos implementados con el fin de hacerlos lo más fidedignos posibles a los modelos originales.

## 6.3 Valoración personal del proyecto

Para acabar el proyecto expondremos una breve valoración personal del mismo.

Nos sentimos satisfechos por los conocimientos obtenidos tras la realización de este proyecto, el cual nos ha hecho ampliar nuestros conocimientos a cerca de las memorias caches y de los protocolos de coherencia más allá de lo visto en las diversas asignaturas de la titulación. Por otra parte, la utilización del simulador y la metodología expuesta en la memoria nos será de gran utilidad a la hora de emprender una carrera en investigación en el área de arquitectura de computadores.

Respecto a la realización del trabajo, creemos que es mejorable en diversos aspectos como las simulaciones realizadas o los modelos implementados, que por falta de tiempo y experiencia con el simulador, no han llegado al nivel esperado. Parte de la falta de tiempo viene derivada de un objetivo inicial de trabajo muy ambicioso: se pretendía simular cada modelo con los 29 benchmarks *SPEC 2k6* y hacer 30 mezclas con estas aplicaciones para los sistemas multiprocesador, cosa que ha sido imposible por los problemas presentados en la sección anterior. Sin embargo, nos sentimos satisfechos por el resultado del mismo en líneas generales.

Por último destacar que se han obtenido conocimientos transversales que no estaban contemplados al inicio del desarrollo del proyecto como pueden ser la automatización de funciones mediante scripts, aprender a trabajar sobre un servidor remoto o la utilización de técnicas como *hints* para comprobar en tiempo de ejecución el correcto funcionamiento del código.

# Referencias


[1] Alexandra Ferrerón, Jesús Alastruey Benedé, Darío Suárez Gracia, Teresa Monreal Arnal, Pablo Ibáñez y Víctor Viñals. 2016. Gestión de contenidos en Caches Operando a Bajo Voltaje.

[2] J.P. Kulkarni, Keejong Kim, and K. Roy, “A 160mV robust schmitt trigger based subthreshold SRAM,” IEEE Journal of Solid-State Circuits, vol. 42, no. 10, pp. 2303– 2313, Oct. 2007.

[3] Shi-Ting Zhou, S. Katariya, H. Ghasemi, S. Draper, and Nam Sung Kim, “Minimizing total area of low-voltage SRAM arrays through joint optimization of cell size, re-dundancy, and ECC,” in IEEE Int. Conf. on Computer Design, 2010, pp. 112–117.

[4] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The gem5 simulator. SIGARCH Comput. Archit. News 39, 2 (August 2011), 1-7. http://gem5.org

[5] Jorge Albericio, Pablo Ibáñez, Víctor Viñals, and Jose María Llabería. 2013. Exploiting reuse locality on inclusive shared last-level caches. *ACM Trans. Archit. Code Optim.* 9, 4, Article 38 (January 2013), 19 pages.

[6] J. L. Baer and W. H. Wang, “On the inclusion properties for multi-level cache herarchies,” in *15th Annual Int. Symp. on Computer Architecture,* 1988, pp. 73-80.

[7] Aamer Jaleel, Eric Borch, Malini Bhandaru, Simon C. Steely Jr., and Joel Emer. 2010. Achieving Non-Inclusive Cache Performance with Inclusive Caches: Temporal Locality Aware (TLA) Cache Management Policies. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture* (MICRO '43). IEEE Computer Society, Washington, DC, USA, 151-162. DOI=http://dx.doi.org/10.1109/MICRO.2010.52

[8] Jorge Albericio, Pablo Ibáñez, Víctor Viñals, and José M. Llabería. 2013. The reuse cache: downsizing the shared last-level cache. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture* (MICRO-46). ACM, New York, NY, USA, 310-321. DOI=http://dx.doi.org/10.1145/2540708.2540735

[9] Arun A. Nair and Lizy K. John. Simulation points for spec cpu 2006. In IEEE International Conference on Computer Design, pages 397–403. IEEE, 2008.

[10] R. Kumar and G. Hinton, "A family of 45nm IA processors," *2009 IEEE International Solid-State Circuits Conference - Digest of Technical Papers*, San Francisco, CA, 2009, pp. 58-59.

[11] H. Lee, S. Cho and B. R. Childers, "Performance of Graceful Degradation for Cache Faults," *IEEE Computer Society Annual Symposium on VLSI (ISVLSI '07)*, Porto Alegre, 2007, pp. 409-415.

[12] John L. Henning, “SPEC CPU2006 benchmark descrip- tions,” SIGARCH Comput. Archit. News, vol. 34, no. 4, pp. 1–17, Sept. 2006.

[13] Z. Chishti, M. D. Powell, and T. N. Vijaykumar. Distance associativity for high-performance energy-e cient non-uniform cache architectures. In Proc. of the 36th annual IEEE/ACM Int. Symp. on Microarchitecture, MICRO 36, pages 55–, Washington, DC, USA, 2003. IEEE Computer Society.

[14] A. Seznec. Decoupled sectored caches: conciliating low tag implementation cost. In Proceedings of the 21st annual international symposium on Computer architecture, ISCA '94, pages 384–393, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[15] Kaxiras, S., Hu, Z., and Martonosi, M. (2001). Cache decay: exploiting generational behavior to reduce cache leakage power. In *Proc. 28th Annual Int Computer Archi- tecture Symp*, pages 240–251

# Anexos

## Anexo 1. Protocolo de coherencia base

En las siguientes secciones se presenta el protocolo de coherencia MESI utilizado como base en detalle.

### Anexo 1.1. Protocolo de coherencia nivel L1

En la fig. 15 se presenta el diagrama completo de estados del protocolo de coherencia, además en la tab.4 se presenta la relación completa entre estados, transiciones y acciones de los mismos.

A continuación se incluye una pequeña leyenda con el significado de cada de uno de los estados — estables en negrita —, transiciones y eventos utilizados en la tabla y el diagrama:

**Estados**:

- **M**: Bloque en lectura y escritura, existente en una única cache L1 y modificado.
- **E**: Bloque en estado exclusivo. Solo lo tiene una cache L1. La diferencia con el estado M es que el bloque se puede leer y escribir pero aún no ha sido escrito.
- **S**: Bloque en solo lectura compartido entre una o más caches L1 y por la L2.
- **NP** / **I**: Bloque inválido.
- IS: Estado transitorio. Se ha efectuado un *GetS* (petición de lectura) y se está esperando su respuesta. El bloque no es legible ni escribible.
- IM: Estado transitorio. Se ha efectuado un *GetX* (petición de escritura) y se está esperando su respuesta. El bloque no es legible ni escribible.
- SM: Estado transitorio. El bloque estaba en estado S y se ha efectuado un *UPGRADE (Write)* (petición de escritura) para conseguir permiso de exclusión. Se está esperando el ACK de todas las caches L1 que compartían ese bloque. El bloque es legible.
- IS_I: Estado transitorio. Mientras la cache L1 está esperando la respuesta de un *GetS* en el estado IS, se recibe una invalidación. Se está esperando el bloque pedido o los ACKs de la invalidación. El bloque no es legible ni escribible.
- M_I: Estado transitorio. Se ha efectuado un remplazo sobre un bloque que estaba en estado M (*PutX*) y se está esperando el WB ACK de la L2.
- SINK_WB_ACK: Estado transitorio. Este estado se alcanza cuando se está esperando el WB ACK del Directorio y se recibe una petición *forward* de otro core.
- PF_IS: Estado transitorio. *GetS* de pre-búsqueda cache no respondido.
- PF_IM: Estado transitorio. *GetX* de pre-búsqueda cache no respondido.
- PF_SM: Estado transitorio. Datos de *GetX* de pre-búsqueda recibido, esperando ACKs.

- PF_IS_I: *GetS* de pre búsqueda efectuado, pero se ha recibido una invalidación del bloque antes que el dato.

**Transiciones**:
- Load: Petición de lectura del procesador.
- Ifetch: Petición de búsqueda del procesador.
- Store: Petición de escritura del procesador.
- Inv: Invalidación de un bloque por L2.
- L1 Replacement: Remplazo de un bloque en L1.
- Fwd. GetX: *GetX* (petición de escritura) de otro procesador.
- Fwd. GetS: *GetS* (petición de lectura) de otro procesador.
- Fwd. Get_Instr: *Get_Instr* (petición de instrucción) de otro procesador.
- Data: Recepción de datos para el procesador.
- Data Excl.: Recepción de datos con permiso exclusivo para el procesador.
- DataS fromL1: Dato de una petición GetS anterior pero con el directorio bloqueado.
- Data Ack: Dato de una petición del procesador desbloqueado.
- Ack: ACK para el procesador.
- Ack all: Último ACK para el procesador.
- WB Ack: ACK de L2 para indicar que el remplazo de un bloque se ha realizado correctamente.
- PF Load: Petición de lectura del sistema de pre-búsqueda.
- PF Ifetch: Petición de búsqueda del sistema de pre-búsqueda.
- PF Store: Petición de escritura del sistema de pre-búsqueda.

**Acciones** (Nombre - Acrónimo):
- a_issueGETS - “a”: Petición de un bloque para lectura a L2.
- pa_issuePfGETS - “pa”: Petición de un bloque para lectura a L2 por el sistema de pre-búsqueda.
- ai_issueGETINSTR - “ai”: Petición de un bloque para lectura de instrucciones a L2.
- pai_issueGETINSTR - “pai”: Petición de un bloque para lectura de instrucciones a L2 por el sistema de pre-búsqueda.
- b_issueGETX - “b”: Petición de un bloque para escritura a L2.
- pb_issuePfGETX - “pb”: Petición de un bloque para escritura a L2 por el sistema de pre-búsqueda.
- c_issueUPGRADE - “c”: Petición de un bloque para escritura a L2 de un bloque que se encuentra en estado S.
- d_sendDataToRequestor - “d”: Envío de la copia local de un bloque a otra L1.
- d2_sendDataToL2 — “d2”: Envío de un bloque a L2.
- dt_sendDataToRequestor_fromTBE - “dt”: Envío de la copia local de un bloque a otra L1 desde el TBE (estructura auxiliar del simulador).

- dt2_sendDataToL2_fromTBE - “d2t”: Envío de la copia local de un bloque a L2 desde el TBE (estructura auxiliar del simulador).
- e_sendAckToRequestor - “e”: Envío del ACK al que efectuó la invalidación (puede ser otra L1 o L2).
- f_sendDataToL2 - “f”: Envío de la copia local de un bloque a L2.
- ft_sendDataToL2_fromTBE - “ft”: Envío de la copia local de un bloque a L2 desde el TBE (estructura auxiliar del simulador).
- fi_sendInvAck - “fi”: Envío del ACK de una invalidación a L2.
- forward_eviction_to_cpu - “\cc”: Envío de información al procesador sobre la eliminación de un bloque.
- g_issuePUTX - “g”: Envío de un bloque a L2 debido a un reemplazo.
- j_sendUnblock - “j”: Envío de *Unblock* sobre un bloque a L2.
- jj_sendExclusiveUnblock - “\j”: Envío de *Unblock Exclusivo* sobre un bloque a L2.
- dg_invalidate_sc - “dg”: Invalidación de un store debido a una condición de carrera.
- h_load_hit - “h”: Notificación de que un load ha sido completado si no es de pre-búsqueda.
- hx_load_hit - “hx”: Notificación de que un load ha sido completado si no es de pre-búsqueda.
- hh_store_hit - “\h”: Notificación de que un store ha sido completado si no es de pre-búsqueda.
- hhx_store_hit - “\hx”: Notificación de que un store ha sido completado si no es de pre-búsqueda.
- i_allocateTBE - “i”: Preparar el TBE para escribir en él y escribir un bloque.
- k_popMandatoryQueue - “k”: Pop de la cola de mensajes *Mandatory.*
- l_popRequestQueue - “l”: Pop de la cola de mensajes *Request.*
- o_popIncomingResponseQueue - “o”: Pop de la cola de mensajes *Incoming Response*.
- s_deallocateTBE - “s”: Vaciar el TBE.
- u_writeDataToL1Cache - “u”: Escribir los datos en la estructura de datos de la cache.
- q_updateAckCount - “q”: Actualizar la cuenta de los ACKs.
- ff_deallocateL1CacheBlock - “\f”: Vaciar un bloque de la cache.
- oo_allocateL1DCacheBlock - “\o”: Modificar el array de tags de la cache de datos con el tag de un bloque.
- pp_allocateL1ICacheBlock - “\p”: Modificar el array de tags de la cache de instrucciones con el tag de un bloque.
- z_stallAndWaitMandatoryQueue - “\z”: Reencolar la petición de la cola *Request.*
- kd_wakeUpDependents - “kd”: Despertar a los dependientes.
- uu_profileInstMiss - “\uim”: Aumentar la cuenta de los fallos en cache de acceso a instrucciones.
- uu_profileInstHit - “\uih”: Aumentar la cuenta de los aciertos en cache de acceso a instrucciones.

- uu_profileDataMiss - “\udm”: Aumentar la cuenta de los fallos en cache de acceso a datos.
- uu_profileDataHit - “\udh”: Aumentar la cuenta de los aciertos en cache de acceso a datos.
- po_observeMiss - “\po”: Informar al sistema de pre-búsqueda del fallo.
- ppm_observePfMiss - “\ppm”: Informar al sistema de pre-búsqueda del fallo parcial.
- pq_popPrefetchQueue - “\pq”: Pop de la cola del sistema de pre-búsqueda.
- mp_markPrefetched - “mp”: Marcar un bloque como pre-buscado.

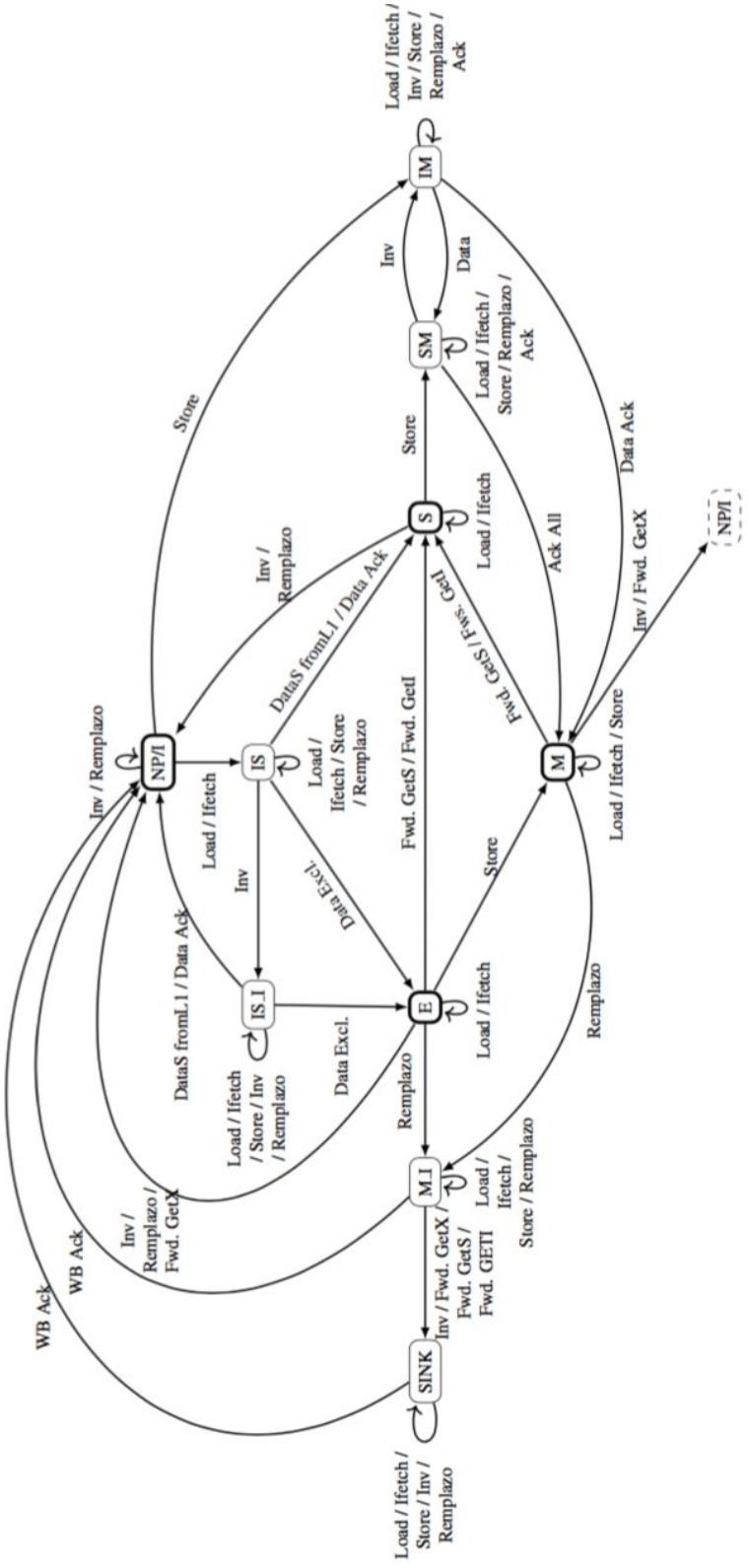



*Figura 15.Diagrama de estados del protocolo en L1.*

| | Load | Ifetch | Store | Inv | Repl. | Fwd. GetX | Fwd. GetS | Fwd GetI | Data | Data Excl. | DataS fromL1 | Data Ack | Ack | Ack all | WB Ack | PF Load | PF Ifetch | PF Store |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| NP | o l a udm po k / IS | p l a l uim po k / IS | o l b udm po k / IM | fi l | f | | | | | | | | | | | o l pa pq /PF_IS | p l pai pq / PF_IS | o l pb pq / PF_IM |
| S | h udh k | h udh k | i c udm k / SM | ce fi l / I | ce fi l / I | | | | | | | | | | | pq | pq | pq |
| E | h udh k | h udh k | ce i g f / M_I | ce fi l / I | ce i g f / M_I | ce d1 / I | d d21 / S | d d21 / S | | | | | | | | pq | pq | pq |
| M | h udh k | h udh k | ce i g f / M_I | ce f l / I | ce i g f / M_I | ce d1 / I | d d21 / S | d d21 / S | | | | | | | | pq | pq | pq |
| IS | z | z | z | fi l / IS_I | z | | | | | u hx j s o kd / E | u j hx s o kd / S | u hx j s o kd / M | q o | | | pq | pq | pq |
| IM | z | z | z | fi l | z | | | | u q o / SM | | | u hx j s o kd / M | q o | | | pq | pq | pq |
| SM | z | z | z | ce fi dg l / IM | z | | | | | | | | qo | j hx s o kd / M | | pq | pq | pq |
| IS_I | z | z | z | fi l | z | | | | | u hxj s o k d / E | u j hx s o kd / I | u hx s o kd / I | | | | pq | pq | pq |
| M_I | z | z | z | fi l / SINK | z | dt d2i1 / SINK | dt d2i1 / SINK | dt d2i1 / SINK | | | | | | | s o kd / I | pq | pq | pq |
| SINK | z | z | z | fi l | z | | | | | | | | | | s o kd / I | pq | pq | pq |
| PF_IS | udm ppm k / IS | udm ppm k / IS | z | fi l / PF_IS_I | z | | | | | u j s mp o kd / E | u j s o kd /S | u s mp o kd / S | | | | pq | pq | pq |
| PF_IM | z | z | udm ppm k / IM | fi l | | | | | u q o / PF_SM | | | u j s mp o kd / M | q o | | | pq | pq | pq |
| PF_SM | z | z | udm ppm k / SM | fi l /PF_IM | z | | | | | | | | q o | j s mp o kd / M | | | | |
| PF_IS_I | udm ppm k / IS_I | | z | fi l | z | | | | | u j s o kd / E | j s o kd / I | s o kd/ I | | | | | | |

*Tabla 4: relación de estados, transiciones y acciones del protocolo en L1.*

## Anexo 1.2. Protocolo de coherencia nivel L2

En la fig. 16 se presenta el diagrama completo de estados del protocolo de coherencia, además en la tab. 5 se presenta la relación completa entre estados, transiciones y acciones de los mismos.

A continuación se incluye una pequeña leyenda con el significado de cada de uno de los estados — estables en negrita —, transiciones y eventos utilizados en la tabla y el diagrama:

**Estados**:

- **NP**: El bloque no está presente en toda la jerarquía cache.
- **SS**: Bloque en solo lectura compartido entre una o más caches L1 y por la L2.
- **M**: Bloque en estado exclusivo. Solo lo tiene la cache L2. Las peticiones de *GetS* y *GetX* de las L1 pueden ser satisfechas directamente por la L2.
- **MT**: Bloque presente en una única cache L1 y con permiso exclusivo.
- M_I: Estado transitorio. La cache está intentando remplazar un bloque y está esperando el WB ACK de memoria. El bloque no es legible ni escribible.
- MT_I: Estado transitorio. La cache está intentando remplazar un bloque en estado MT. Se ha enviado la correspondiente invalidación a la cache L1 correspondiente y se está esperando el ACK. El bloque no es legible ni escribible.
- MCT_I: Estado transitorio. Es igual que MT_I pero se sabe que el bloque está limpio en la cache L2.
- I_I: Estado transitorio. La L2 cache está intentando remplazar un bloque en estado SS y limpio. Se han enviado las correspondientes invalidaciones a las caches L1 y se están esperando los ACKs. El dato no es legible ni escribible.
- S_I: Estado transitorio. Es igual que I_I pero el bloque está sucio por lo que el bloque tiene que ser enviado a memoria. El dato no es legible ni escribible.
- ISS: Estado transitorio. La cache L2 ha recibido un *GetS* de una cache L1 sobre un bloque que no está presente en la jerarquía cache. Se ha enviado la petición de lectura a memoria y se está esperando su respuesta.
- IS: Estado transitorio. Es igual que ISS excepto que el bloque pedido es de instrucciones o el bloque lo pide más de un core.
- IM: Estado transitorio. La cache L2 ha recibido un *GetX* de una cache L1 que no está presente en la jerarquía cache. Se ha enviado la petición a memoria y se está esperando su respuesta.
- SS_MB: Estado transitorio bloqueante. Se llega cuando una cache L1 pide un bloque con exclusividad y el bloque estaba en estado SS. Esto significa que había copias legibles de este bloque en caches L1. Antes de dar el permiso exclusivo a la cache L1, estas copias tienen que ser invalidadas. En este estado las invalidaciones han sido enviadas y la cache L1 que ha pedido el bloque en exclusividad ha sido avisada con el número de ACKs que le tienen que llegar. Una

vez que ésta reciba todos los ACKs responderá a la L2 cache con *Unblock* para que pueda satisfacer otras peticiones sobre este bloque.

- MT_MB: Estado transitorio bloqueante. Se llega cuando la cache L2 ha enviado un bloque con exclusividad a una cache L1 que lo ha pedido pero aún no se ha recibido el *Unblock*.
- MT_IIB: Estado transitorio bloqueante. Se llega cuando la cache L2 recibe una petición *GetS* sobre un bloque con estado exclusivo por otra cache L1 (estado MT). Entonces, la L2 cache hace un *forward* de la petición a la correspondiente cache L1 y cae en este estado. Tienen que pasar dos eventos antes que se pueda desbloquear este bloque: La cache L1 con permiso exclusivo del bloque tiene que enviar una copia a la cache L2 para actualizar el valor y la cache L1 que ha pedido el bloque tiene que enviar *Unblock* a la cache L2 como señal de que ha recibido el bloque.
- MT_IB: Estado transitorio bloqueante. Se llega cuando en el estado MT_IIB la cache L2 recibe el *Unblock* de la cache L1 que ha pedido el bloque pero falta por recibir la copia de *Write-Back* de la cache L1 que poseía el bloque.
- MT_SB: Estado transitorio bloqueante. Se llega cuando en el estado MT_IIB la cache L2 recibe la copia de *Write-Back* de la cache L1 que poseía el bloque pero falta por recibir el *Unblock* de la cache L1 que ha pedido el bloque.

**Transiciones**:

- L1_GET_INSTR: Petición de lectura de una cache L1 sobre un bloque de instrucciones.
- L1_GETS: Petición de lectura de una cache L1 sobre un bloque de datos.
- L1_GETX: Petición de escritura de una cache L1 sobre un bloque de datos.
- L1_UPGRADE: Petición de escritura de una cache L1 sobre un bloque de datos el cual ya poseía y estaba en estado S.
- L1_PUTX: Remplazo de un bloque por una cache L1.
- L1_PUTX_old: Remplazo de un bloque por una cache L1 y no existen más caches L1 que posean ese bloque.
- L2_Replacement: Remplazo de un bloque en la cache L2.
- L2_Replacement_clean: Remplazo de un bloque limpio en la cache L2.
- Mem_Data: Recepción de datos de memoria.
- Mem_Ack: ACK de memoria.
- WB_Data: Recepción de un bloque de datos de una cache L1 debido a una invalidación.
- WB_Data_clean: Recepción de un bloque de datos limpio de una cache L1 debido a una invalidación.
- Ack: ACK de una cache L1.
- Ack_all: Último ACK de los que se estaban esperando de una cache L1.
- Unblock: Desbloqueo por parte de la cache L1 que había solicitado un bloque.
- Exclusive_Unblock: Desbloqueo por parte de una cache L1 que había solicitado un bloque con permiso exclusivo.

**Acciones** (Nombre - Acrónimo):

- a_issueFetchToMemory - “a”: Pedir bloque a memoria.
- b_forwardRequestToExclusive - “b”: Pedir el bloque a la cache L1 que lo tiene en estado exclusivo.
- c_exclusiveReplacement - “c”: Enviar bloque a memoria debido a un remplazo.
- c_exclusiveCleanReplacement - “cc”: Enviar ACK a memoria de remplazo de un bloque limpio.
- ct_exclusiveReplacementFromTBE - “ct”: Enviar bloque a memoria desde el TBE.
- d_sendDataToRequestor - “d”: Enviar bloque a la cache L1 que lo ha pedido.
- dd_sendExclusiveDataToRequestor - “dd”: Enviar bloque con permiso exclusivo a la cache L1 que lo ha pedido.
- ds_sendSharedDataToRequestor - “ds”: Enviar bloque compartido a la cache L1 que lo ha pedido.
- e_sendDataToGetSRequestors - “e”: Enviar bloque a todas las caches L1 que lo han pedido (*GetS*).
- ex_sendExclusiveDataToGetSRequestors - “ex”: Enviar bloque con permiso exclusivo a todas las caches L1 que lo han pedido (*GetS*).
- ee_sendDataToGetXRequestor - “ee”: Enviar bloque a la cache L1 que lo ha pedido (*GetX*).
- f_sendInvToSharers - “f”: Invalidar a las caches L1 que poseen un bloque que va a ser remplazado por la cache L2.
- fw_sendFwdInvToSharers - “fw”: Invalidar a las caches L1 que comparten un bloque debido a una petición de otra cache L1.
- fwm_sendFwdInvToSharersMinusRequestor - “fwm”: Invalidar a las caches L1 que comparten un bloque menos a una que es la que lo ha pedido.
- i_allocateTBE - “i”: Escribir un bloque en el TBE para la petición.
- s_deallocateTBE - “s”: Borrar el bloque que estaba en el TBE.
- jj_popL1RequestQueue - “\j”: Pop de la cola de peticiones de las caches L1.
- k_popUnblockQueue - “k”: Pop de la cola Unblock.
- o_popIncomingResponseQueue - “o”: Pop de la cola Incoming Response.
- m_writeDataToCache - “m”: Escribir el bloque de la cola de respuestas a la cache.
- mr_writeDataToCacheFromRequest - “mr”: Escribir el bloque de la cola de respuestas a la cache.
- q_updateAck - “q”: Actualizar la cuenta de los ACKs.
- qq_writeDataToTBE - “\qq”: Escribir el bloque de la cola de respuestas al TBE.
- ss_recordGetSL1ID - “\s”: Guardar la ID de la L1 cache que ha efectuado la petición *GetS* para la correspondiente respuesta.
- xx_recordGetXL1ID - “\x”: Guardar la ID de la L1 cache que ha efectuado la petición *GetX* para la correspondiente respuesta.
- set_setMRU - “\set”: Actualizar el MRU del correspondiente bloque.
- qq_allocateL2CacheBlock - “\q”: Modificar el array de tags de la cache con el tag de un bloque.

- rr_deallocateL2CacheBlock - “\r”: Modificar el array de tags de la cache borrando el tag de un bloque.
- t_sendWBAck - “t”: Enviar WB ACK.
- ts_sendInvAckToUpgrader - “ts”: Enviar ACK de invalidación a la cache L1 que ha efectuado un *Upgrade.*
- uu_profileMiss - “\um”: Aumentar la cuenta de los fallos en cache.
- uu_profileHit - “\uh”: Aumentar la cuenta de los aciertos en cache.
- nn_addSharer - “\n”: Añadir una cache L1 a la lista de *Sharers* de un bloque.
- nnu_addSharerFromUnblock - “\nu”: Añadir una cache L1 a la lista de *Sharers* de un bloque debido a un *Unblock.*
- kk_removeRequestSharer - “\k”: Quitar una cache L1 de la lista de *Sharers* de un bloque.
- ll_clearSharers - “\l”: Borrar la lista de *Sharers* de un bloque.
- mm_markExclusive - “\m”: Actualizar el dueño exclusivo de un bloque.
- mmu_markExclusiveFromUnblock - “\mu”: Actualizar el dueño exclusivo de un bloque.
- zz_stallAndWaitL1RequestQueue - “zz”: Reencolar la petición de la cola de peticiones de las caches L1.
- zn_recycleResponseNetwork - “zn”: Reencolar la petición de la memoria.
- kd_wakeUpDependents - “kd”: Despertar a los dependientes.

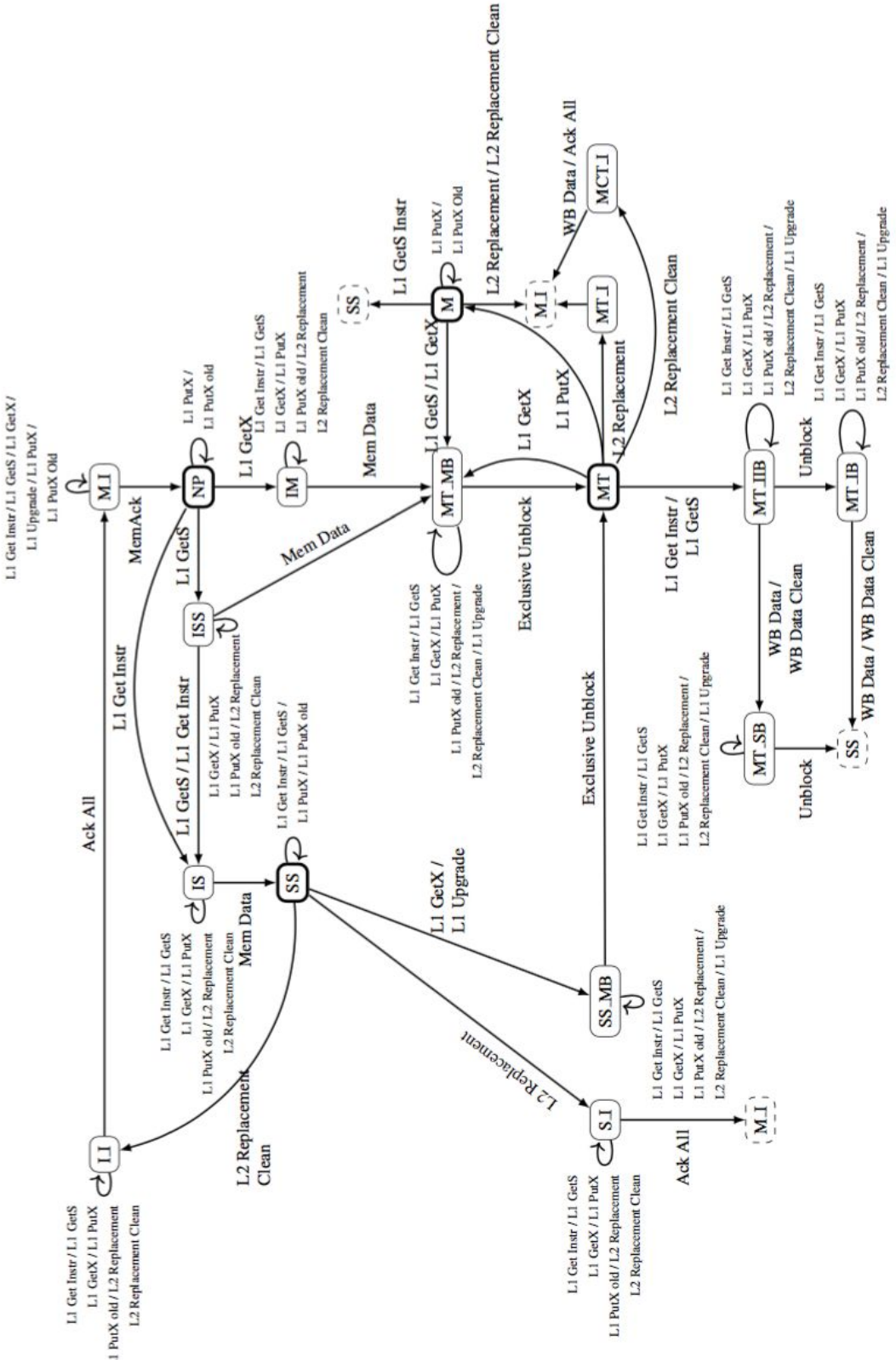



*Figura 16. Diagrama de estados del protocolo de coherencia en L2.*

| | L1 Get Inse | L1 GetS | L1 GetX | L1 Up-grade | L1 PutX | L1 PutX Old | Reemplazo | Reemplazo Clean | Mem Data | Mem Ack | WB Ack | WB Data Clean | Ack | Ack All | Unblock | Exclusive Un-block | Mem Inv |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| NP | q j n i s a um j / IS | q j n i s a um j /ISS | q j i x a um j / IM | | i j | i j | | | | | | | | | | | o |
| SS | ds n set uh j | ds n set uh j | d fwm set uh j / SS_MB | fwm ts set uh jj / SS_MB | i j | i j | i f r /SJ | i f r /LJ | | | | | | | | | i f r /SJ |
| M | d n set uh j /SS | dd set uh j / MT_MB | d set uh j / MT_MB | | i j | i j | i c r / MJ | i ce r / MJ | | | | | | | | | i f r / MTJ |
| MT | h um set j / MT_IIB | h um set j / MT_IIB | h um set j / MT_MB | | l mr i j / M | i j | i f r / MTJ | i f r / MCTJ | | | | | | | | | i f r / MTJ |
| MJ | zz | zz | zz | zz | i j | i j | | | | s o kd / NP | | | | | | | o |
| MTJ | zz | zz | zz | zz | zz | zz | | | | | qq ct o / MJ | cc o / MJ | | ct o / MJ | | | o |
| MCTJ | zz | zz | zz | zz | zz | zz | | | | | qq ct o / MJ | cc o / MJ | | cc o / MJ | | | o |
| LJ | zz | zz | zz | zz | i j | i j | | | | | | | q o | cc o / MJ | | | o |
| SLJ | zz | zz | zz | zz | i j | i j | | | | | | | q o | ct o / MJ | | | o |
| ISS | n s um j / IS | n s um j / IS | zz | | i j | i j | zz | zz | m es s o kd / MT_MB | | | | | | | | m |
| IS | n s um j | n s um j | zz | | i j | i j | zz | zz | m e s o kd / SS | | | | | | | | m |
| IM | zz | zz | zz | | i j | i j | zz | zz | m ee s o kd / MT_MB | | | | | | | | m |
| SS_MB | zz | zz | zz | zz | zz | zz | zz | zz | | | | | | | mu k kd / MT | | m |
| MT_MB | zz | zz | zz | zz | zz | zz | zz | zz | | | | | | | mu k kd / MT | | m |
| MT_IIB | zz | zz | zz | zz | zz | zz | zz | zz | | | m o / MT_SB | m o / MT_SB | | | mu k kd / MT | | m |
| MT_IB | zz | zz | zz | zz | i j | i j | zz | zz | | | m o kd / SS | m o kd / SS | | | | | m |
| MT_SB | zz | zz | zz | zz | i j | i j | zz | zz | | | | | | | mu k kd / SS | | m |

*Tabla 5. Relación de estados, transiciones y acciones del protocolo de coherencia de L2.*

## Anexo 1.3. Directorio

En la, fig 17, se presenta el diagrama de estados del directorio, además en la tabla 6 se presenta la relación completa de acciones y transiciones.

A continuación se incluye una pequeña leyenda con el significado de cada de uno de los estados, estables en negrita, transiciones y eventos utilizados en la tabla y el diagrama:

- Estados:
  - **I**: el directorio es el propietario del bloque y la memoria tiene la versión más reciente. El resto de copias están invalidadas.
  - **M**: el bloque está en las memorias caches, y puede que la versión de memoria no esté actualizada.
  - ID: espera del dato de memoria por una lectura del DMA.
  - ID_W: espera del ACK de memoria por una escritura del DMA.
  - IM: espera del dato de memoria por una petición de un bloque de la memoria cache.
  - M: espera del ACK de memoria por el envió de un dato recibido de la memoria cache.
  - MDRD: espera del dato por parte de una lectura del DMA.
  - MDRDI: espera del ACK de memoria por una lectura del DMA.
  - MDWR: espera del dato por parte de una escritura del DMA.
  - MDWRI: espera del ACK de memoria por parte de una escritura del DMA.
- Transiciones:
  - Fetch: búsqueda de un bloque a memoria.
  - Data: recepción del writeback de datos de memoria.
  - Memory Data: llegada de datos de memoria buscados.
  - DMA Read: petición de lectura de memoria al DMA.
  - DMA Write: petición de escritura de memoria al DMA.
  - Clean Replacement: Reemplazo de un bloque limpio de L2.
- Acciones:
  - Queue off-chip fetch request --- qf: encolar petición de búsqueda hacia memoria principal.
  - Pop incoming request queue --- j: eliminamos la petición de la pila.
  - Recycle request queue --- z: reencolar la petición.
  - Invalidate --- inv: invalidación de un bloque de cache.
  - Queue off-chip writeback request --- qw: encolar petición de writeback hacia memoria principal.
  - Pop incoming request queue --- k: eliminamos la petición de la cola.
  - Send Data to DMA --- dpr: envío de datos al controlador DMA por una petición PutX.
  - Queue off-chip writeback --- qwf: encolar una petición de writeback.

- Send Data to DMA --- dr: envio de un dato al DMA desde el directorio.
- Memory Data --- d: llegada de una petición de búsqueda desde memoria.
- Pop off-chip request --- q: elimar de la pila una petición de memoria principal.
- Wake-up dependents --- kd: despertar a acciones dependientes.
- Send Ack to DMA --- da: envío de ACK a la controladora de DMA.
- Send Ack --- aa: envío de ack a L2.
- Deallocate TBE --- w: expulsión del bloque del TBE
- Recycle DMA --- zz: reencolar la petición del DMA.
- Allocate TBE --- a: inclusión del bloque en el TBE.

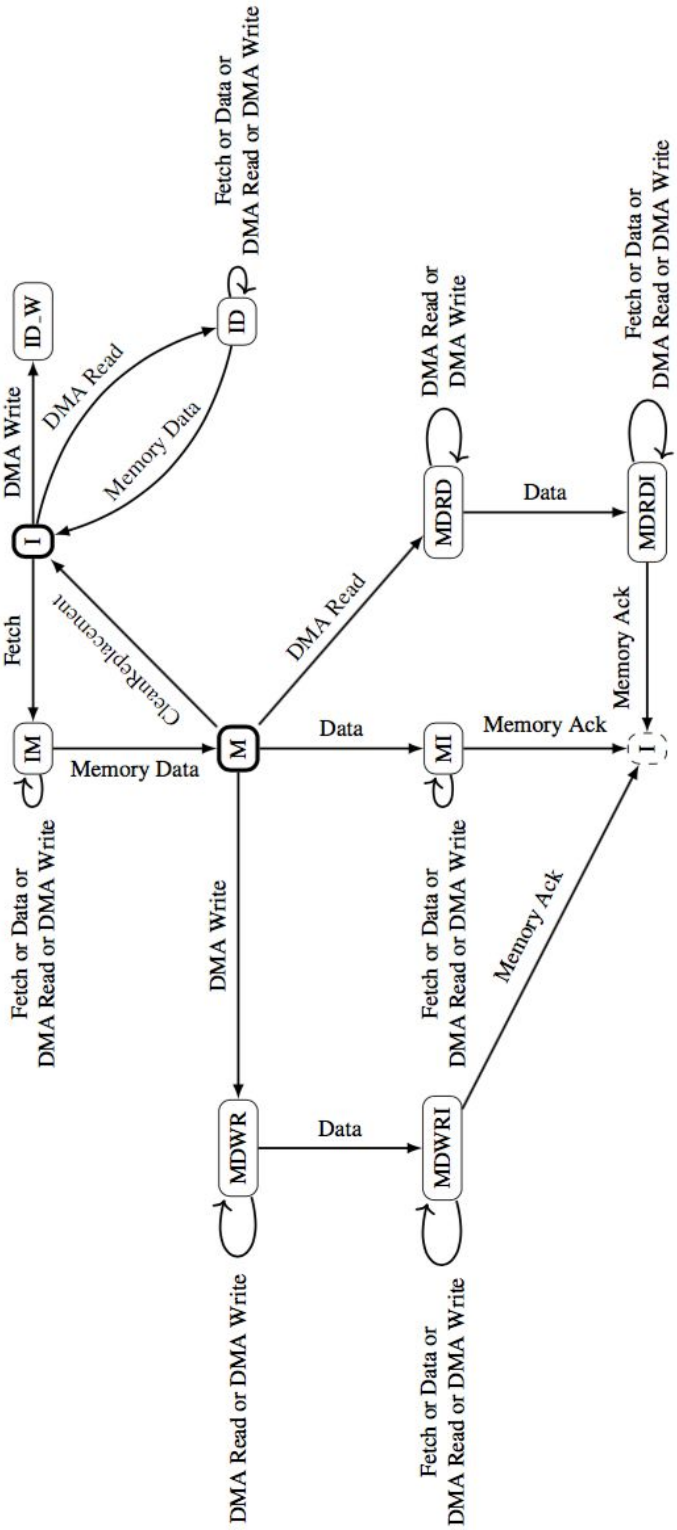



*Figura 17. Diagrama de estados para el directorio.*

| | Fetch | Data | Memory Data | Memory Ack | DMA Read | DMA Write | Clean Replace-ment |
|---|---|---|---|---|---|---|---|
| I | qf j / IM | | | | qfd j / ID | qwp j / ID_W | |
| ID | z | z | dr q kd / I | | zz | zz | |
| ID_W | z | z | | da q kd / I | zz | zz | |
| M | inv z | qw k / MI | | | inv j / MDRD | v inv j / MDWR | a k kd / I |
| IM | z | z | d q kd / M | | zz | zz | |
| MI | z | z | | aa q kd / I | zz | zz | |
| MDRD | | drp qw k / MRDDI | | | zz | zz | |
| MDRDI | z | z | | aa q kd / I | zz | zz | |
| MDWR | | qwt k / MDWRI | | | zz | zz | |
| MDWRI | z | z | | aa da w q kd / I | zz | zz | |

*Tabla 6: relación de estados, transiciones y acciones del directorio de coherencia.*

# Anexo 2. BDOT

En el siguiente anexo se presentan los cambios realizados al protocolo de coherencia para conseguir implementar el BDOT con reemplazo específico. Para ello se han incluido nuevos estados transitorios y transiciones, ver fig 19.

- Estados
  - WD_SS: LLC espera la llegada de un bloque debido a un intercambio producido en estado SS.
  - WD_MT: LLC espera la llegada de un bloque debido a un intercambio producido en estado MT.
  - WD_M: LLC espera la llegada de un bloque debido a un intercambio producido en estado M.
- Transiciones:
  - L1_PutX_N: Notificación del reemplazo de la última copia de un bloque en cache privada a LLC.
  - L1_GetS_T: Petición de un bloque T para lectura.
  - Swap_init: Iniciación de un intercambio entre dos bloques.
  - L1_GetS_Instr: petición de un bloque T de instrucciones para lectura.
  - L1_PutX_T: Notificación del reemplazo de un bloque T en la cache privada.

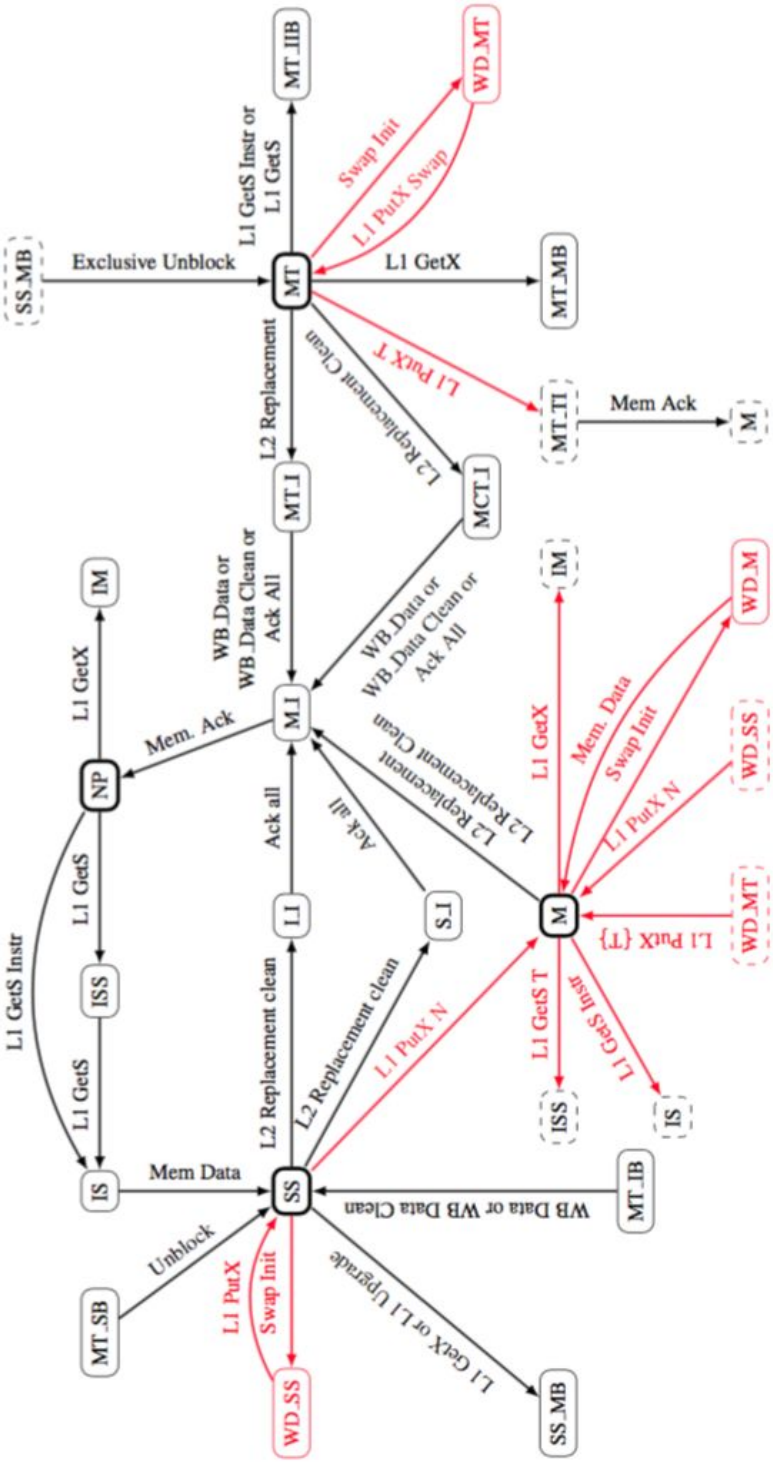



*Figura 18. Diagrama del protocolo de coherencia en L2 con BDOT.*

## Anexo 2.1 Directorio

La inclusión del BDOT en la LLC requiere también la modificación del directorio para que este sea capaz de pedir bloques a memoria principal que se encuentran en la LLC. Para ello se ha incluido un nuevo estado TMI: espera del ACK de memoria tras el envío de un bloque a memoria de una L2 cuya entrada para ese bloque es de tipo T; y dos nuevas transiciones: Data Tag: envío de un bloque a memoria principal de L2 cuya entrada es tipo T; y Fetch T: búsqueda de un bloque a memoria principal por una petición en L2 cuya entrada es de tipo T, ver fig. 19.

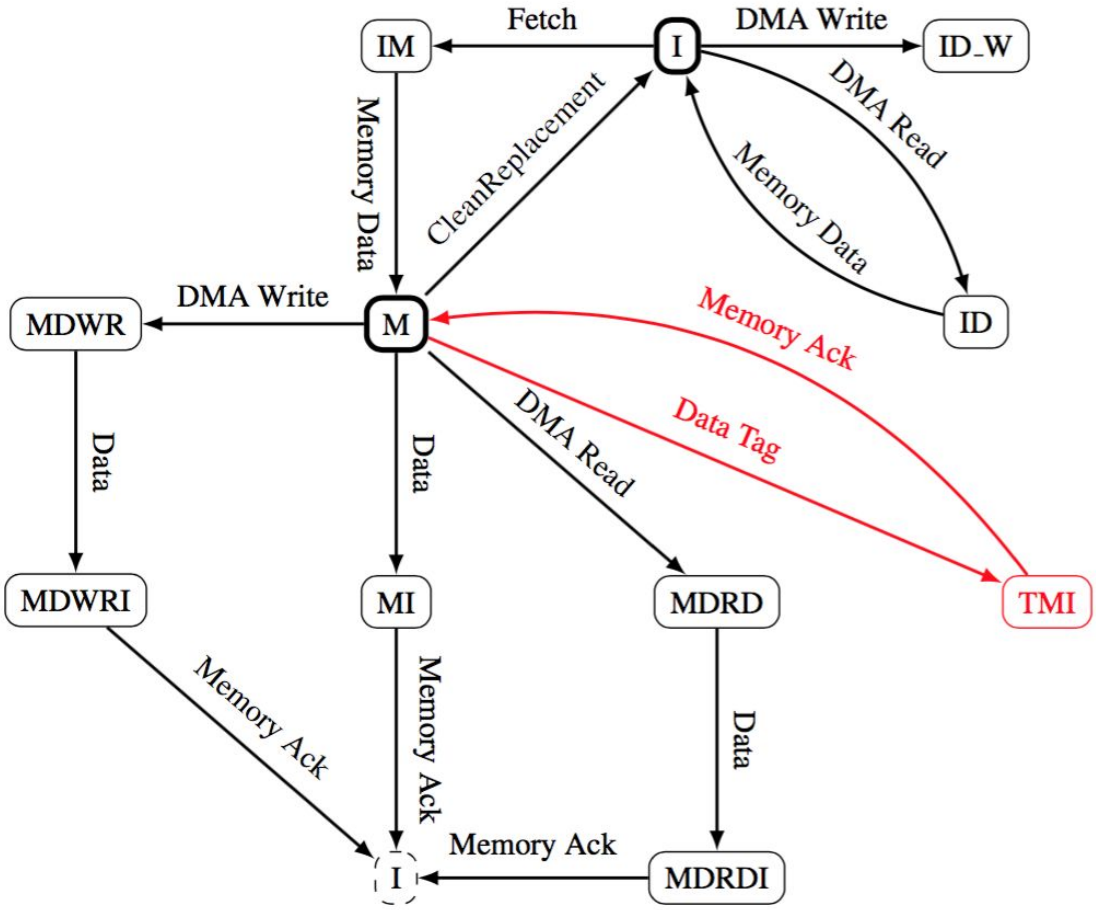



*Figura 19. Diagrama de estados del directorio.*

# Anexo 3. Manual de utilización de *Gem5*

Como parte del trabajo realizado, se ha elaborado un manual en el cual se recogen los pasos necesarios para instalar *Gem5* en sobre Ubuntu 12 y una breve introducción al simulador.

## Anexo 3.1. Instalación sobre Ubuntu 12

1.- Resolver las dependencias

```
$> sudo apt-get update
$> sudo apt-get upgrade
$> sudo apt-get install mercurial scons swig gcc m4 python python-dev libgoogle-perftools-dev
g++
```

2.- Descargar los fuentes del repositorio remoto. En este caso, se descarga la versión estable.

```
$> hg clone http://repo.gem5.org/gem5-stable
```

3.- Obtener versión de gcc ≥ 4.7, en este caso, la versión 4.8.

```
$> sudo apt-get install python-software-properties
$> sudo add-apt-repository ppa:ubuntu-toolchain-r/test
$> sudo apt-get update
$> sudo apt-get install gcc-4.8
$> sudo update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-4.8 50
```

4.- Compilar el simulador. En esta guía se trabaja con la versión *.opt* de la arquitectura *ALPHA*.

```
gem5/$> scons build/ALPHA/gem5.opt PROTOCOL=MOESI_CMP_directory RUBY=True
SLICC_HTML=True -j8
```

Para ejecutar programas en el simulador, en la línea de comandos hay que seguir la siguiente estructura:

```
gem5/$> <gem5 binary> [gem5 options] <simulation script> [script options]
```

5.- Ejecutar el programa de prueba 'hello world'.

```
gem5/$> build/ALPHA/gem5.opt configs/example/se.py -c
tests/test-progs/hello/bin/alpha/linux/hello
```

6.- Ejecutar el programa de prueba 'ruby_random_test.py'.

```
gem5/$> build/ALPHA/gem5.opt configs/example/ruby_random_test.py
```

Para ver las estadísticas generadas por el simulador hay que acceder a */m5out/stats.txt*

## Anexo 3.2. Estructura del simulador

A continuación se presenta las carpetas más importantes del simulador, con una breve descripción de lo que contienen:

- **configs:** scripts de configuración para la ejecución del simulador.
- **src:** código fuente del proyecto
  - **arch:** implementaciones de la arquitectura
  - **base:** estructuras de datos generales
  - **cpu:** modelos de cpu
  - **dev:** modelos de dispositivos
  - **mem:** subsistema de memoria
    - **cache**: implementación del modelo clásico de memoria.
    - **ruby**: implementación del modelo ruby de memoria.
    - **protocol:** definición del protocolo de coherencia.

## Anexo 3.3. Modificar algoritmo de reemplazo

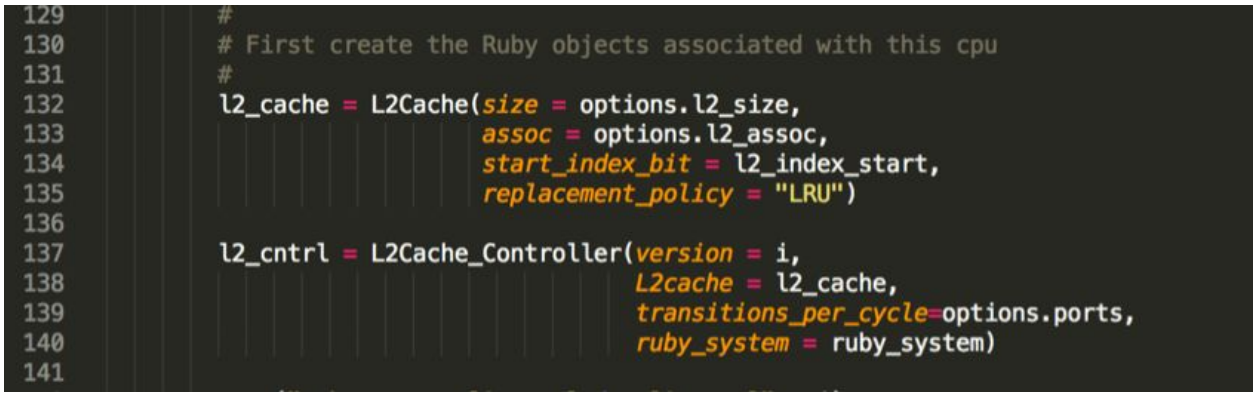
A continuación, trabajando con el protocolo de memoria MESI de dos niveles, se propone cambiar el algoritmo de reemplazo por defecto (pseudoLRU) de la caché de primer nivel y de segundo nivel por LRU.

1.- La implementación de ambos algoritmos (pseudoLRU y LRU) se encuentra en:
*gem5/src/mem/ruby/structures/PseudoLRUPolicy.hh*
*gem5/src/mem/ruby/structures/LRUPolicy.hh*

2.- Para cambiar los algoritmos de reemplazo que son usados por el simulador hay que añadir las líneas 83, 88 y 135 del fichero *gem5/configs/ruby/MESI_Two_Level.py* de la manera que aparece en las siguientes figuras 19 y 20.

```
72      #
73      l2_bits = int(math.log(options.num_l2caches, 2))
74      block_size_bits = int(math.log(options.cacheline_size, 2))
75
76      for i in xrange(options.num_cpus):
77          #
78          # First create the Ruby objects associated with this cpu
79          #
80          l1i_cache = L1Cache(size = options.l1i_size,
81                              assoc = options.l1i_assoc,
82                              start_index_bit = block_size_bits,
83                              replacement_policy = "LRU",
84                              is_icache = True)
85          l1d_cache = L1Cache(size = options.l1d_size,
86                              assoc = options.l1d_assoc,
87                              start_index_bit = block_size_bits,
88                              replacement_policy = "LRU",
89                              is_icache = False)
90
```

*Figura 20. LRU en cache L1.*



```
129         #
130         # First create the Ruby objects associated with this cpu
131         #
132         l2_cache = L2Cache(size = options.l2_size,
133                            assoc = options.l2_assoc,
134                            start_index_bit = l2_index_start,
135                            replacement_policy = "LRU")
136
137         l2_cntrl = L2Cache_Controller(version = i,
138                                       L2cache = l2_cache,
139                                       transitions_per_cycle=options.ports,
140                                       ruby_system = ruby_system)
141
```

*Figura 21. LRU en cache L2.*

3.- Se compila.

gem5/$> scons build/ALPHA/gem5.opt PROTOCOL=MESI_Two_Level RUBY=True SLICC_HTML=True -j8

4.- Y por último, se ejecuta.

gem5/$> build/ALPHA/gem5.opt configs/example/ruby_random_test.py

## Anexo 3.4. Nuevo algoritmo de reemplazo

A continuación se especifican los pasos para la creación de un neuvo algoritmo de remplazo.

1.- Crear fichero nuevo <*.hh> en /src/ruby/mem/structures.
2.- Heredar de la clase *AbstractReplacementPolicy.hh*

3.- Include del fichero en CacheMemory.hh

```
#include "mem/ruby/structures/NRRPolicy.hh";
```

4.- Añadir las siguientes líneas en CacheMemory.cc para que sea posible crear el objeto:

```
else if (m_policy == "NRR")
        m_replacementPolicy_ptr =
                new NRR_BDPolicy(m_cache_num_sets, m_cache_assoc);
```

5.- Cambiar el fichero *gem5/configs/ruby/MESI_Two_Level.py* para utilizar la nueva política.

```
l2_cache = L2Cache(size = options.l2_size,
          assoc = options.l2_assoc,
          start_index_bit = l2_index_start,
          replacement_policy = "NRR")
```

## Anexo 3.5. Ficheros Slicc

A continuación se muestran los fichero más importantes del MESI con su función:

- **MSI_Two_Level-L1cache.sm:** nivel L1 del protocolo.
- **MSI_Two_Level-L2cache.sm:** nivel L2 del protocolo.
- **MSI_Two_Level-dir.sm:** nivel de directorio del protocolo.
- **MSI_Two_Level-msg.sm:** mensajes intercambiados entre los niveles del protocolo.
- **RubySlicc_Types:** definición de tipos y estructuras usadas en el protocolo.
- **RubySlicc_Exports:** definición de tipos y estructuras externas usadas en el protocolo.

## Anexo 3.6. Instalación del compilador cruzado de ALPHA

Para poder realizar checkpoints en modo SE como se explica en este apartado hay que tener instalado un compilador de ALPHA. La wiki oficial de Gem5 provee uno en el siguiente enlace: http://www.m5sim.org/dist/current/alphaev67-unknown-linux-gnu.tar.bz2 válido para x86 Linux (64 bits).

Para su instalación hay que descomprimir el fichero descargado y añadir su *bin* a la variable de entorno PATH.

## Anexo 3.7. Creación de checkpoints

Para este ejemplo de creación de checkpoints se va a utilizar un programa en C que resuelve el problema de las 8 reinas:

1. Descargamos el programa mediante el siguiente comando: *$>wget https://llvm.org/svn/llvm-project/test-suite/tags/RELEASE_14/SingleSource/Benchmarks/McGill/queens.c*

2. Añadimos en el código del programa lo siguiente:
   a. La cabecera *#include "util/m5/m5op.h"*
   b. En el punto del programa donde queremos que se realice el checkpoint añadimos la siguiente instrucción: *m5_checkpoint(0,0);*
   c. En el punto en que queremos que finalice la ejecución del programa añadimos la instrucción: *m5_exit(0);*

3. Compilamos nuestro programa con el siguiente comando:

   *$> alphaev67-unknown-linux-gnu-gcc -DUNIX -o queens-w-checkpoint queens.c ../util/m5/m5op_alpha.S --static*

4. Ejecutamos el simulador con el programa, con el detalle mínimo para acelerar la misma: *$> ./build/ALPHA_Hammer/gem5.opt configs/example/se.py –c ./queens-w-checkpoint –o 8 –cpu-type=AtomicSimpleCPU.* Es obligatorio utilizar en Ruby el modelo Hammer debido a que es el único que tiene caches con operación *flush* (eso dice la wiki).

## Anexo 3.8. Ejecución a partir de un checkpoint

Para la ejecución de un programa a partir de un checkpoint hay que ejecutar el siguiente comando:

*$> ./build/ALPHA/gem5.opt configs/example/se.py –c ./queens –o 8 --restore-with-cpu=timing --ruby ---cpu-type=detailed --checkpoint-dir=m5out –r 1*

## Anexo 3.9. Uso de Simpoints

A continuación se describen los pasos necesarios para realizar simpoints y checkpoints a partir de los mismos:

1.- Descargar los fuentes de Simpoint (Simpoint 3.2)
$> wget https://cseweb.ucsd.edu/~calder/simpoint/releases/SimPoint.3.2.tar.gz

2.- Descomprimir los fuentes
$> tar -xzvf SimPoint.3.2.tar.gz

3.- Antes de compilar es necesario añadir los siguientes includes:
En el fichero *Utilities.h:*
#inlcude <cstdlib>, #include <limits.h>

En el fichero *CmdLineParser.h:*
#inlcude <cstdlib>

En el fichero *Datapoint.h:*
#include <iostream>, #include <limits.h>, #include <stdlib.h>

En el fichero *FVParser.h:*
#inlcude <string.h>

4.- Añadir la ruta a la carpeta */bin* de simpoint a la variable de entorno PATH

5.- Generar el fichero BBV file.
$> ./build/ALPHA_MESI_Two_Level/gem5.opt ./configs/example/se.py -c ./benchmark/queens -o 12 --simpoint-profile --fastmem --cpu-type=AtomicSimpleCPU

6.- El análisis del fichero BBV se realiza mediante el comando:

$> simpoint -loadFVFile m5out/simpoint.bb.gz -maxK 30 -saveSimpoints m5out/ss -saveSimpointWeights m5out/sw -inputVectorsGzipped

7.- Tomamos los checkpoints a partir de los ficheros generados.

$> . /build/ALPHA_MOESI_hammer/gem5.opt ./configs/example/se.py --take-simpoint-checkpoint=m5out/ss,m5out/sw,10000000,30000 --cpu-type=TimingSimpleCPU -c ./benchmark/queens -o 12

8.- Podemos ejecutar el programa a partir de los checkpoints mediante el siguiente comando:

$>./build/ALPHA_MESI_Two_Level/gem5.opt ./configs/example/se.py --restore-simpoint-checkpoint -r 1 --checkpoint-dir m5out/ --cpu-type=detailed --ruby --restore-with-cpu=timing -c ./benchmark/queens -o 12