Universitat Politècnica de València

Escola Tècnica Superior d'Enginyeria Informàtica

Proyecto Final de Carrera:

Simulación de nuevas arquitecturas de memorias cache de procesadores para sistemas empotrados.

Para la obtención del título de

Ingeniero en Informática

Autor :
Carlos Catalá Barber

Dirigida por:
Jose Vicente Busquets Mataix

Valencia, Septiembre de 2011

Abstract

Nowadays, many embedded processors include in their architecture on-chip static memories, so called scratch-pad memories (SPM), either coexisting of replacing cache memories. Compared to cache, these memories do not require tags and complex control logic, thus resulting in increased efficiency both in silicon area and energy consumption. Last years, many papers have proposed some algorithms to carefully allocate memory segments in SPM in order to enhance performance and/or to reduce energy requirements. However, very few, slightly care about the SPM architecture itself, to make it more controllable, more power efficient and faster. In this paper, we propose a new control paradigm for the SPM to update it contents on the fly. Our solution is based on a small control unit to automatically load code into the SPM whilst it is fetched for execution. We extend the processor architecture with a few new instructions to control the SPM, and introduce different execution modes. The resulting architecture reduces the SPM updating delays, which motivates a very dynamic use of the SPM (i.e. frequently updating its contents during program execution). This technique is orthogonal and complementary to many of the solutions presented to date to efficiently use the SPM. We test our proposal in a derivation of the Simplescalar simulator, with typical embedded benchmarks. The results show improvements, on average, of 30.6% in energy saving and 7.6% in performance compared to a system with cache

Keywords: Computer architecture, Simplescalar, Vatios, Cache memory, Scracthpad memory, Computer simulation, energy consumption

Palabras clave: Arquitectura de computadores, Simplescalar, Vatios, Memoria Cache, Memoria Scracthpad, Simulación de computadores, Consumo energía

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

CAPÍTULO 1 : RESUMEN

Actualmente mas del 95% de los procesadores fabricados se montan en sistemas empotrados. Muchos de estos procesadores se montan en dispositivos móviles alimentados por baterías o sistemas de tiempo real donde un bajo consumo de energía puede ser extremadamente necesario. Gran parte del gasto energético de un procesador es consumido por las memorias on-chip, esto hace que cobre un especial interés la reducción energética de estas memorias sin que ello conlleve una reducción de las prestaciones en dichos procesadores.

Actualmente, muchos procesadores empotrados incluyen en su arquitecturas memorias estáticas on-chip llamadas scratch-pad memories (SPM), coexistiendo o remplazando a las memorias cache. Comparadas con la cache estas memorias no requieren de etiquetas y una compleja lógica de control lo que conlleva un incremento en la eficiencia tanto en el área de silicio gastada como en el consumo energético. En los últimos años muchos estudios han propuesto algunos algoritmos para meter cuidadosamente segmentos de memoria en la SPM para incrementar el rendimiento y/o reducir el consumo de memoria. Sin embargo muy poco han cambiado la arquitectura de la SPM para hacerla mas controlable, mas eficiente energéticamente y más rápida.

En esta memoria presentamos tres posibles técnicas para mejorar el rendimiento y/o consumo energético en un procesador empotrado con una cache convencional. La primera de ella consiste en introducir y bloquear trozos de código en la propia memoria cache, lo que resulta bastante útil en sistemas de tiempo real ya que permite ajustar la cota del WCET repercutiendo en un mejor aprovechamiento del procesador, en la segunda sustituimos la memoria cache por una spm y por ultimo en la tercera de estas técnicas proponemos un nuevo paradigma de control de la SPM para actualizar sus contenido al vuelo, mediante diversos cambios hardware y software.

Esta ultima solución esta basada en una pequeña unidad de control que carga código en la SPM mientras este es lanzado a ejecución. Nosotros extendemos la arquitectura del procesador con unas pocas nuevas instrucciones para controlar la SPM, y añadimos diferentes modos de ejecución. La arquitectura resultante reduce los retrasos por la actualización de código en la SPM y motiva a un uso muy dinámico de esta, es decir con actualizaciones de su contenido frecuentes durante la ejecución del programa. Esta técnica presentada es una técnica ortogonal que puede complementarse con diversas técnicas presentadas hasta la fecha para el eficiente uso de la SPM.

Todas estas técnicas han sido implementadas en un simulador basado en el popular Simplescalar y han mostrado mejoras en los resultados, de media, de un 30,6% de mejora en el consumo energético y un 7,6% en el rendimiento de la ultima técnica implementada respecto una sistema convencional con cache.

CAPÍTULO 2 : INTRODUCCIÓN

En los últimos años, la creciente popularidad comercial de los dispositivos móviles, con procesadores empotrados en su interior, como móviles ,PDAs, cámaras, reproductores de MP4, etc. han atraído un elevado interés económico. Como consecuencia, han sido muchos los estudios realizados para incrementar el rendimientos computacional de estos dispositivos a fin de poderles incorporar la mayor funcionalidad posible. Sin embargo este incremento del rendimiento no se ha visto acompañado de un incremento en la capacidad de las baterías. A pesar del gran avance dado gracias a las baterías de ion de litio, el avance en la capacidad de las baterías avanza lentamente y un mayor consumo energético requiere de una batería de mayor tamaño. Consecuentemente, mientras que la tecnología de las baterías avanza lentamente se debe hacer un esfuerzo para reducir el consumo energético de estos dispositivos móviles.

Comparados con los PCs de propósito general, este tipo de dispositivos suelen tener una carga de trabajo prácticamente fija y conocida cuando el sistema es diseñado. Esta importante característica debe ser explotada, y son muchas las técnicas empleadas para meter trozos de código o datos en la memoria on-chip (cache, spm ...) del procesador a fin de reducir el gasto energético.
En computadoras de propósito general, las memorias caches han jugado un rol decisivo en proveer el ancho de memoria requerido a los procesadores. De hecho, esta es una de las técnicas mas importantes para reducir el famoso cuello de botella en la memoria. Las memorias caches tienen un comportamiento muy dinámico e impredecible, capaz de adaptar su contenido a cualquier carga de trabajo. Sin embargo no son eficientes desde el punto de vista del consumo energético debido a que para ello requieren dos componentes adicionales como son la memoria de etiquetas y la lógica de comparación. Algunos estudios han empezado a identificar al subsistema de memoria como el cuello de botella energético de todo el sistema.

El gran consumo energético de la memoria cache, y la predecible carga de trabajo en los sistemas empotrados, han contribuido a que las SPM emerjan como una una alternativa eficiente a estas memorias. Ademas de ser mas eficientes energéticamente hablando, su total predictibilidad, juega un importante rol en sistemas de tiempo real. Sus principales desventajas son derivadas del hecho de que la SPM es básicamente una pequeña y rápida memoria mapeada dentro del espacio de direcciones de la memoria principal. Por lo tanto, sus operaciones deben ser explicitamente mapeadas por el enlazador (linker), o por programación. En la ultima década se han presentados muchas aproximaciones para seleccionar cuidadosamente los contenidos que deben ser guardados en la SPM. Algunos de estos estudios han considerado que el contenido de la SPM debía ser constante durante la totalidad del tiempo de ejecución del programa (aproximación estática), mientras que otros han considerado que como mejor, el cambio de los contenidos de manera dinámica para adecuar el contenido de la SPM a puntos calientes del programa (secciones de código que se ejecutan frecuentemente) durante la ejecución del mismo. Algunos estudios también han presentados distintas modificaciones en la arquitectura de la SPM para mejorar el control sobre esta memoria.

Por otra parte en sistemas empotrados de tiempo real todos los componentes hardware que tienen una latencia variable y no conocida en tiempo de compilación, como la memoria cache, presentan problemas a la hora de calcular el WCET. El WCET, calculo del peor tiempo de ejecución de una tarea, infrautiliza los recursos del sistema en aras de tener la seguridad de que las tareas se ejecutan antes de que se cumpla su tiempo limite, por lo que un calculo demasiado pesimista de este tiempo puede repercutir en un menor rendimiento del sistema. Es por esto que en este tipo de sistemas se

suelen utilizar técnicas como el bloqueo de los contenidos de la memoria cache o memorias totalmente predecibles en tiempos de compilación como las SPM que ayudan al que el calculo del WCET sea mas sencillo y ajustado.

En esta memoria presentamos y estudiamos dos de estas técnicas de arquitecturas de memorias on-chip en sistemas empotrados, ademas de presentar una nueva y original técnica para mejorar el control y la actualización de contenidos en una memoria SPM. Esta nueva técnica esta enfocada a reducir lo máximo posible el tiempo de sobrecarga resultante de la actualización de los contenidos de la SPM.

Este hecho permitirá introducir técnicas para adaptar dinámicamente el contenido de la SPM en tiempo de ejecución con un coste de retardo reducido. Estas técnicas favorecerán las actualizaciones frecuentes, para ser capaces de contener los puntos calientes de un programa en un momento dado.

La nueva técnica implementada en esta memoria es totalmente ortogonal y complementaria a muchas de las técnicas ya propuestas de selección cuidadosa de contenidos para ser cargados en la SPM. De hecho nuestra solución incrementa los beneficios de dichas técnicas simplemente adoptando las extensiones en la arquitectura del procesador propuestas.

Esta técnica esta implementada mediante una serie de pequeños cambios en la arquitectura del procesador. Se han añadido algunas nuevas instrucciones para controlar los contenidos en la SPM. Haciendo cambios en el modo de ejecución del procesador, la CPU puede actualizar dinámicamente en "vuelo" el contenido de la SPM con el código lanzado de la memoria principal a ejecución. Usualmente el código seleccionado para ser metido en la SPM proviene de bucles o cuerpos de funciones. También hemos añadido un pequeño bucle de instrucciones para beneficiar la localidad espacial del código que no se guarda dentro de la SPM.

El principal objetivo de esta propuesta es la reducción del consumo energético quedando en segundo plano el rendimiento del procesador. También se ha puesto un especial interés en que la solución resultante sea realista en el sentido de que sea simple y fácil de usar. Es importante destacar que se ha optado por una solución que pueda ser aplicada en el mundo real siendo mas prioritario la simplicidad de esta que la obtención de un rendimiento impresionante. Otros estudios incluyen en sus soluciones unidades funcionales adicionales como pueden ser MMUs o DMAs, que pueden llegar a ser demasiado complejos para algunos sistemas empotrados sin embargo nuestra propuesta puede ser implementada en todo el espectro de sistemas empotrados y no solo en los de gama mas alta.

Las propuestas presentadas están enfocadas solo a secciones de código. La carga de trabajo en sistemas empotrados es usualmente predecible y menor, en comparación con sus datos. Por lo tanto, el código puede acumular un gran número de pequeños puntos calientes, que contribuirán a aprovechar al máximo el pequeño tamaño de estas memorias onchip.

En los siguientes capítulos repasaremos el estado del arte de los sistemas de memoria on-chip. Veremos conceptos teóricos sobre los sistemas empotrados, arquitecturas de computadores y su subsitemas de memoria, necesarios para entender la totalidad de esta memoria. A continuación veremos las distintas herramientas utilizadas en este proyecto, y por último veremos los tres sistemas de memorias on-chip implementados en esta memoria y los resultados obtenidos de la simulación de estos sistemas de memoria con distintos benchmarks.

2.1 - RELATED WORKS

Existen una gran cantidad de estudios sobre la problemática que presenta el uso de memorias caches en sistemas empotrados de tiempo real. En [36] L.C. Aparicio hace una retrospectiva sobre los distintos métodos utilizados para predecir el comportamiento de la memoria cache y así calcular una cota del WCET mas ajustada, haciendo especial hincapié en el bloqueo de contenidos en la memoria cache. Esta técnica modifica el comportamiento natural de la memoria cache impidiendo la inclusión de nuevo contenido en ella mientras esta bloqueada, lo que hace que gane en determinismo aunque ello conlleve una perdida de prestaciones. Es fácilmente implementable ya que muchos de los procesadores actuales permiten dicho bloqueo. En [37] Martí, Perez, Perles y Busquets proponen la utilización de un algoritmo genético para seleccionar el conjunto de instrucciones ,que minimizan el tiempo de ejecución de las tareas, que deben ser bloqueadas en la cache. En [38] Martí amplia dicho estudio haciendo un análisis mas exhaustivo. En este estudio Martí presenta dos tipos de técnicas para el bloqueo de contenidos en la memoria cache, una técnica estática donde los contenidos se cargan en la cache al principio de la ejecución y permanecen invariables durante toda la ejecución del programa y una técnica dinámica donde los contenidos no permanecen fijos durante todo el funcionamiento del sistema sino que en determinados instantes se cambia su contenido y estos permanecen fijos hasta que se llega a un nuevo instante de cambio de contenidos. Ambas técnicas están destinadas a mejorar el calculo del WCET permitiendo una mejor utilización del procesador en sistemas multitareas de tiempo real. En ambos casos utiliza algoritmos genéticos para la selección de contenidos obteniendo así una selección subóptima de contenidos en un tiempo de ejecución aceptable.

Puaut en [39] y [40] también trabaja con memorias caches con bloqueo de contenidos. En el primer articulo se marca como objetivo determinar si un sistema multitarea no planificable con memorias caches no deterministas, podría ser planificable con memorias caches con los contenidos bloqueados. Mientras que en el segundo articulo presenta dos algoritmos de baja complejidad para la selección de contenidos a bloquear en la memoria cache. El primero de ellos se marca como objetivo la minimización del tiempo de ejecución necesario para ejecutar todas las tareas, mientras que el segundo algoritmo se intenta minimizar las interferencias entre las tareas.

Todos los estudios presentados anteriormente se centran únicamente en las memorias caches de instrucciones, debido en gran parte a que los accesos a memoria para la búsqueda de instrucciones representan prácticamente el 75% de todos los accesos a memoria referenciados por la cpu lo que hace que cobre un mayor interés el estudio de cache de instrucciones sobre la de datos. De todas maneras también hay estudios sobre la aplicación de diversas técnicas de bloqueo de cache sobre la cache de datos, por ejemplo, Puaut en [41] extiende uno de sus algoritmos de baja complejidad para trabajar tanto con datos como con instrucciones.

En otro orden de cosas, también son muchos los estudios acerca de la mejora en el consumo energético y/o en el tiempo de ejecución que supone la utilización eficiente de una spm en lugar de una cache. Todos estos estudios presentan una serie de técnicas sobre la asignación de código en la spm que pueden dividirse en dos tipos, las que realizan un aproximación estática, los contenidos de la spm son asignados de antemano y permanecen invariables durante la ejecución del programa, y las que realizan una aproximación dinámica en las cuales los contenidos de la spm cambian en tiempo de ejecución.

[1], [2], y [3] presentan técnicas de asignación de objetos de memoria en la spm estática, donde es

necesario conocer en tiempo de compilación el tamaño de la spm. Banakar et al. en [1] utiliza un knapsack algorithm en tiempo de compilación para decidir de forma estática los datos o código que deben estar dentro de la scratchpad. Verma et al. en [2] utiliza un grafo de conflictos entre los distintos bloques básicos de memoria para mediante la formulación de un problema de programación lineal entera obtener el conjunto de objetos de memoria que debe contener la scratchpad. En [3] Verma et al. mejora esta propuesta para un entorno multitarea con tres posibles estrategias de partición del espacio de la scratchpad entre los procesos implicados.

En [4] Nguyen et al. presenta también una técnica de asignación estática, pero a diferencia de las anteriores no es necesario conocer el tamaño de la spm en tiempo de compilación, ya que esta técnica retrasa la decisión de que objetos de memoria deben estar o no en la scratchpad hasta la carga de la aplicación, esto posible ya que esta técnica guarda alguna información de profiling dentro del binario de la aplicación.

Egger et al. en [5] [6] y [7], Hyungmin Cho et al. en [8] , Janapsatyat et al. en [9], Steinke et al. en [10], Polleti et al [11] , Lian Li et al. en [13] y Doosan et al. [19] presentan técnicas dinámicas de asignación de objetos de memoria en la scrathpad. En [5] Egger et al. presenta una técnica donde los bucles son introducidos en la spm por demanda, y el mejor conjunto de bucles para ser copiados en la scratchpad son elegidos mediante la resolución de un problema de programación lineal entera. Egger et al. en [6] propone una serie de cambios hardware en la MMU para el manejo de la scratchpad, permitiendo la carga en la scrathpad de los trozos de código mas frecuentemente utilizados bajo demanda. Hyungmin Cho et al. en [8] se basa en esta técnica pero la utiliza para cargar datos en la scratchpad. Egger et al. en [7] mejora la técnica presentada en [6] mediante la utilización de un bit en cada entrada de la TLB llamado SPM flag. En [9] Janapsatyat et al. introduce una serie de cambios hardware-software para guardar en la spm los trozos de código mas utilizados. Se introducen una serie de instrucciones especiales en tiempo de compilación en diversos puntos clave mediante un algoritmo heurístico, que activan un controlador hardware que se encarga de manejar el flujo de datos de la scratchpad, las principales ventajas de nuestra nueva solución respecto a esta técnica es que nuestra propuesta no requiere que el tamaño de la spm sea conocido en tiempo de compilación, que nuestra propuesta requiere un menor numero de instrucciones y menos lógica de control para funcionar. Steinke et al. en [10] nos presenta una técnica donde los bloques de código son copiados en la scratchpad antes de su ejecución, el conjunto de instrucciones optimo a copiar son determinados mediante la resolución de un problema de programación lineal entera. En [11] Poletti et al. propone en su trabajo una serie de modificaciones hardware software para un sistema multitarea donde los objetos son movidos de la memoria principal a la scratchpad con la ayuda de un API creado para tal fin. En este paper Poletti et al. utiliza una DMA junto a la scratchpad para reducir el coste de copiar datos de memoria principal a la spm, ademas de una manera semejante a nuestra solución utiliza un interfaz de programación de alto nivel para manejar las trasferencias de datos mediante la DMA, la principal diferencia con nuestra nueva propuesta consiste en el mayor coste monetario y gasto energético de esta aproximación por la utilización de la DMA. En [12] Lian Li et al. presenta una técnica para la inserción de arrays de datos en la spm. Utilizando grafos de conflictos de rangos de vida, un graph-coloring algorithm (conflict graph of live ranges, a graph-coloring algorithm) determina que arrays de datos y en que instante deben copiarse a la scratchpad.

Doosan Cho en [19] et al. propone una serie de cambios hardware y software, especialmente diseñados para el manejo de una spm en aplicaciones multimedia, con el fin de reducir el consumo energético. Este tipo de aplicaciones tienen un comportamiento impredecible hasta su ejecución lo

que hace que sean difícil de optimizar estáticamente por el compilador. Su propuesta consiste en un algoritmo ,usado en tiempo de compilación, basado en profiling para seleccionar una serie de escenarios de datos (data layout scenacios). Estos son seleccionados y optimizados por una rutina , runtime SPM manager, que es introducida en el sistema operativo y que se ejecuta periódicamente. Esta rutina gracias al análisis hecho en tiempo de compilación y a un nuevo componente hardware donde guarda una historia de los accesos a memoria, cargan escenarios de datos (layout data scenarios) en la SPM con la ayuda de una DMA.
Lee en [13] presenta un esquema de memoria on-chip basado en la utilización ,junto a la cache de primer nivel , de una pequeña memoria on-chip llamada loop cache cuya finalidad es reducir el consumo energético en el lanzamiento de instrucciones. Utiliza esta memoria para meter bucles en ella mediante una serie de instrucciones especiales que le permiten interactuar con el controlador de dicha memoria.

Nosotros en esta memoria presentamos una nueva técnica ortogonal que puede complementarse con cualquier algoritmo de asignación de código a la spm de los propuestos por los artículos anteriores. Nuestra propuesta se basa en una serie de de pequeños cambios hardware-software para permitir la copia de instrucciones de la memoria principal a la scratchpad. Presentamos una serie de instrucciones de alto nivel que permiten mover partes del código, funciones y bucles, a la scratchpad de forma sencilla y dinámica quedando de momento en manos del programador la elección de los segmentos de código a copiar.

CAPÍTULO 3 : SISTEMAS EMPOTRADOS

Un sistema empotrado es una combinación hardware y software, y quizá partes adicionales mecánicas o electrónicas, diseñado específicamente para realizar unas pocas funcionas, a menudo solamente una. Frecuentemente se usan como parte de un sistema mas amplio como un microondas, un automóvil o una central nuclear.

Suele estar compuesto en su parte central por una CPU que añade capacidad de computo al sistema, ya sea mediante un microprocesador o un microcontrolador. Aunque en ocasiones es posible construir un dispositivo con las misma funcionalidad remplazando el procesador y el software por un circuito integrado. Sin embargo este diseño hardware no se suele utilizar ya que el diseño del sistema empotrado mediante un procesador y software ofrece una mayor flexibilidad, es mas barato, mas fácil de diseñar y tiene un menor gasto energético.

Este tipo de sistemas suelen funcionar sin que el usuario final deba tener constancia de la existencia del procesador o del software ejecutado. Por lo general el software tiene una misión fija y especifica a la aplicación.

Uno de los principales objetivos de estos sistemas es la reducción de costes. Los sistemas empotrados suelen usar un procesador relativamente pequeño y una memoria pequeña para reducir costes. En general se suele recurrir a arquitecturas de procesadores simples dejando normalmente de lado arquitecturas superescalares que si que se utilizan en los microprocesadores usados en Pcs o servidores.

Muchos de estos sistemas se emplean en dispositivos portátiles donde cobra especial importancia el bajo consumo energético, ya que todos estos sistemas deben ser alimentados mediante baterías y un bajo consumo energético repercutirá en una mayor autonomía del dispositivo. También es de especial interés que posean un peso bajo y unas pequeñas dimensiones.

Otra características común que la mayoría de sistemas empotrado requieren es la necesidad de interactuar con su entorno mediante diversos tipos de dispositivos de entrada salida especiales, lo que puede requerir un acondicionamiento de las diferentes señales.

Ademas también la robustez es otra característica que puede ser importante en este tipo de sistemas ya que muchos de estos dispositivos están embarcados en sistemas transportados sujetos a vibraciones o incluso a impactos, ademas que muchos requieren trabajar en condiciones no óptimas de temperatura humedad o limpieza.

Este tipo de sistemas debe ser fiable y seguro ante los errores, ya que puede requerir un comportamiento autónomo. Es fundamental asegurar que si el sistema de control falla el sistema controlado se quede en un estado estable y seguro.

El primer sistema empotrado moderno reconocido fue el sistema de guía de la misión Apollo. Desde entonces han sido utilizados para multitud de fines, militares, médicos, aeroespaciales, en la industria del automóvil, ect... Algunos de los sistemas empotrados que podemos encontrar en nuestra vida cotidiana son:

- Equipos de redes, routers, switches, etc...

- Electrodomésticos como lavadoras, microondas, lavavajillas, etc...
- Electrónica de consumo como reproductores de MP3, teléfonos móviles, PDA's, cámaras de fotos digitales, etc...

La mayoría de estos sistemas tienen requisitos de tiempo real.

3.1 - SISTEMAS DE TIEMPO REAL

Tal como hemos comentado muchos de los sistemas empotrados tienen requisitos de tiempo real, es decir deben responder en un determinado espacio de tiempo a eventos externos, ejecutando la tarea correspondiente antes de que acabe el tiempo limite. Podemos definir un sistema de tiempo real como un sistema informático que realiza sus funciones y responde a eventos externos asíncronos en un tiempo acotado. Es decir que no basta con que realice correctamente sus funciones sino que ademas debe hacerlo en un intervalo de tiempo determinado. Estos sistemas pueden dividirse en dos tipos principalmente, los sistemas de tiempo real estrictos (hard real-time systems) y sistemas de tiempo real flexibles (soft real-time systems).

Los sistemas de tiempo real estrictos son aquellos en los cuales el retraso de los resultados puede resultar desastroso, pudiendo provocar graves daños materiales y humanos, incluyendo la destrucción del propio sistema e incluso el daño o perdida de vidas humanas. Estos sistemas se utilizan para controlar sistemas críticos y es extremadamente fundamental que todas las acciones terminen dentro del plazo determinado. Se utilizan en aviónica como por ejemplo en el sistema de control de vuelo, en la industria aeroespacial, en la industria del automóvil, como en el control de frenado del ABS, etc....

Mientras que los sistemas de tiempo real flexibles son aquellos en que un retraso en la obtención de resultados no provoca grandes fallos críticos mas allá una degradación en el rendimiento del sistema. En ese tipo de sistemas es deseable que se cumplan los plazos de todas las tareas pero ocasionalmente se pueden perder plazos sin que el resultado reporte ningún tipo de catástrofe. Este tipo de sistemas se utilizan en sistemas multimedia, redes de computadores, sistemas de reconocimiento de voz, sistemas de adquisición de datos etc....

Hay que resaltar que aunque la mayoría de sistemas de tiempo real se puedan englobar en uno o otro de estos dos tipos, puede darse el caso de que en un mismo sistema convivan tareas de ambos requisitos temporales, siendo necesario establecer cierta prioridad a las tareas para que el sistema funcione correctamente.

Para poder cumplir todos estos requisitos temporales y que las tareas se ejecuten en su plazo previsto, garantizando el correcto funcionamiento del sistema, se deberá realizar un estudio de las tareas y el hardware donde se ejecutan estas. Este estudio toma el nombre de análisis de planificabilidad.

Todo programa que se ejecuta en el sistema ,compartiendo los recursos de este, viene definida por cuatro valores (C, D, T, φ). El valor C se conoce como tiempo de computo. Es el tiempo de CPU necesario para completar la tarea en cada una de las activaciones. Este tiempo puede variar en cada activación debido a que su ejecución en la CPU depende de elementos hardware no deterministas y de latencia variable como por ejemplo la memoria cache, el predictor de saltos, pipeline del procesador, y también del propio software. Para que la planificación del sistema sea segura, todas

las tareas deben terminar su ejecución en su tiempo de computo establecido y para ello debemos escoger siempre el peor escenario posible. Y es por esto que en el análisis del sistema siempre se calcula el tiempo de ejecución en el peor caso o WCET (*worst case execution time)* y se considera que el tiempo de computo de la tarea es siempre igual a su WCET.
Calcular el WCET es una tarea complicada , y es realmente importante en sistemas de tiempo real. El calculo de una cota de WCET extremadamente pesimista y alejada del tiempo de computo real del sistema puede llevar, en muchos caso, a que los test de planificabilidad indiquen que un sistema no es planificable, o que el sistema se muestre como planificable pero a costa de infrautilizar los recursos de este.

El valor D se refiere al tiempo de finalización o deadline. Este valor hace referencia al tiempo máximo que puede transcurrir entre la activación de tarea y su finalización u obtención de resultados. Este tiempo depende únicamente del sistema físico que se pretende controlar siendo independiente del sistema en que se esta ejecutando la tarea.

El valor T hace referencia al periodo entre cada activación de una tarea, según su periodo las tareas se pueden clasificar en varias categorías:

-Tareas periódicas: Son aquellas tareas que se ejecutan en intervalos regulares. Entre cada activación de la tarea pasan exactamente T unidades de tiempo.

-Tareas aperiódicas: Son tareas que suelen responden a eventos externos. Se pueden clasificar dos tipos. Esporádicas, tareas que no se conoce el instante cuando se activara la tarea, pero si se conoce su frecuencia máxima, o su periodo mínimo es decir tendrán una separación mínima entre activaciones de T unidades de tiempo . O estocásticas, tareas que no se conoce cuando se activaran ni en que frecuencia lo harán.

Por último el valor φ hace referencia al retardo de inicio. Es el tiempo que tarda la tarea entre la activación de la tarea hasta que esta empieza a ser ejecutada. Este retardo viene influido por la granularidad del reloj, la ejecución del planificador, la decisión del planificador sobre que tarea debe ejecutarse, o adicionalmente por la disponibilidad de los datos que necesita la tarea para ejecutarse.

Todas las tareas del sistema de tiempo real se ejecutan compartiendo los recursos del mismo, y es el algoritmo planificador el que se encarga de meter o sacar de ejecución a las tareas, según sus restricciones temporales o prioridades. Hay varios tipos de planificadores, se pueden dividir en estáticos y dinámicos.

Los planificadores estáticos analizan los requisitos temporales antes de la ejecución de todas las tareas del sistema y determinan entonces el orden en que se ejecutaran. Se basan en la construcción de una tabla de tiempos con tamaño igual al hiperperiodo, mínimo común múltiplo de los periodos todas las tareas del sistema. Una vez lanzado el sistema el planificador lo único que tiene que hacer es seguir dicha tabla, donde están indicados los instantes que se debe activar cada tarea. Este proceso es cíclico y se repite una vez acabado el hiperperiodo de las tareas. Las ventajas de este método residen en la poca sobrecarga que introducen en el sistema a ejecutar, sin embargo este tipo de planificadores es poco flexible y difícil de diseñar en sistemas complejos lo que reduce su uso en la actualidad a los sistemas mas sencillos.

Los planificadores dinámicos son aquellos en que el orden de las tareas no se conoce antes de lanzar el sistema, sino que se va decidiendo durante la ejecución de este. El orden de las tareas se va decidiendo según su periodo o prioridad. En este tipo de planificadores se les puede asignar a las tareas una prioridad, para dar mas urgencia o importancia a una tarea, y el sistema siempre ejecutara la tarea lista con una mayor prioridad. Si la prioridad se le asigna en la fase de diseño del sistema y permanece invariable durante la ejecución del mismo se dice que es un sistema con prioridades fijas, mientras que si esta varia durante la ejecución del sistema se le denomina de prioridades dinámicas. También se pueden clasificar según su capacidad para cambiar la tarea que actualmente posee la cpu por otra sin que esta haya finalizado su ejecución. A este tipo de planificadores se les denomina expulsivos.

En la actualidad los algoritmos de planificación mas utilizados son los planificadores dinámicos con prioridades fijas en sistemas con expulsión de tareas, ya que son los que mejor comportamiento presentan, sobretodo en caso de sobrecarga. En este tipo de sistemas las tareas con mas prioridad entran en ejecución una vez están listas, si hace falta, expulsando a la que se esta ejecutando en ese momento, lo que hace que en caso de sobrecarga sean las tareas que menor prioridad tienen las que pierdan su plazo.

3.2 - ARQUITECTURA HARDWARE EN SISTEMAS EMPOTRADOS

En este apartado y sus correspondientes subapartados definiremos las principales características hardware de todo sistema empotrado para disponer de una visión general de dichos dispositivos.

Entendemos como hardware a todo componente físico y tangible que forma una computadora. Es decir todos los componentes eléctricos, electrónicos, mecánicos y cualquier otro tipo de componente físico presente en un sistema.

Podemos dividir la parte hardware de todo sistema informático en tres partes. La CPU, encargada de procesar los datos, el subsistema de memoria, encargada de guardar y suministrar datos e instrucciones a la CPU , y los dispositivos de entrada y salida, dispositivos que permiten la introducción y salida de datos de una computadora.

3.2.1 - CPU

La central processing unit (CPU) o unidad central de proceso es la parte encargada de ejecutar las instrucciones y procesar los datos que conforman los programas. Esta compuesta principalmente por varios registros, una unidad de control y una o varias unidades aritmético lógicas.

Forman parte de microprocesadores fabricados mediante complejas técnicas litográficas en obleas de silicio de un tamaño fijo que actualmente ronda los 300 mm^2, y comparten espacio en cada microprocesador con otros elementos como pueden ser el coprocesador matemático o la memoria on-chip, que puede ser una memoria cache o una spm. Estas obleas de silicio contienen varios cientos de microprocesadores, una vez terminado todo el proceso litográfico, son cortados y cada uno de las plaquitas resultantes es dotada de una capsula protectora normalmente de plástico y conectadas a cientos de pines. Dado la importancia del coste por chip en los sistemas empotrados hay que destacar que cuantos más microprocesadores por oblea se puedan fabricar, más barato nos resultara el microprocesador ya que lo realmente costoso es litografiar la oblea no el coste de los materiales utilizados que en estos casos es despreciable. Por lo tanto es de fundamental importancia

el coste espacial en silicio del microprocesador en este tipo de sistemas y es por ello que algunas de las técnicas mas complejas utilizadas en CPUs para uso personal o servidores no son utilizadas en este tipo de sistemas.

La ejecución de toda instrucción en una CPU sigue siempre las mismas pautas, primero la CPU busca en memoria la siguiente instrucción a ejecutar, a continuación decodifica dicha instrucción determinando el tipo de instrucción y lo que debe hacer con ella para que se ejecute correctamente. A continuación busca en los registros los operandos para dicha instrucción, en el caso de que fueran necesarios. Una vez hecho la unidad aritmético lógica (ALU) sera la encargada de operar sobre los datos preparados. Por último se guardaran los resultados en un registro o memoria principal según la instrucción ejecutada.

En las primeras CPU se realizaban todas estas operaciones en un solo ciclo de reloj. En el afán mejorar el rendimiento de la CPU se procedió a segmentar estas etapas, mejorando así las prestaciones de esta mediante el paralelismo en la ejecución de instrucciones, donde varias instrucciones pueden estar ejecutándose en paralelo en la CPU cada una en una de sus etapas.

Gracias a la segmentación al hacer las etapas mas pequeñas podemos disminuir el tiempo de ciclo aumentando la velocidad de la CPU. El objetivo de la segmentación es solapar la ejecución de las instrucciones de manera semejante a una cadena de montaje de manera que en cada ciclo de reloj cada instrucción completa una parte de su ejecución y pasa a la siguiente fase. Una vez la instrucción ha recorrido y ejecutado todas las fases finaliza su ejecución. Idealmente en cada ciclo de reloj siempre hay una instrucción que finaliza su ejecución por lo que podemos decir que en condiciones ideales y con cada una de las etapas con una duración perfectamente equilibrada, el tiempo por instrucción de la maquina segmentada sera igual:

$$\frac{\text{Tiempo por instrucción en la máquina no segmentada}}{\text{Número de etapas de la segmentación}}$$

Sin embargo, es habitual que las etapas no estén perfectamente equilibradas, además la segmentación también conlleva cierto gasto. Por lo que el tiempo por instrucción no tendrá su mínimo valor posible aunque pueda estar próximo.

Las etapas típicas en que podemos dividir todo procesador segmentado son:

- Instruction Fetch (IF) : Búsqueda de la instrucción.
- Instruction Decoder (ID): Decodificación de la instrucción y búsqueda de operandos.
- Execute (EX): Ejecución y cálculo de direcciones efectivas.
- Memory access (MEM): Acceso a memoria.
- Write Back (WB): Almacenamiento del resultado.

En las dos siguientes figuras, la figura 1 y la figura 2, podemos observar una comparación entre la ejecución de instrucciones en un procesador sin segmentación y un procesador con sus etapas

segmentadas. La figura 1 muestra la ejecución de instrucciones en un procesador sin sus etapas segmentadas. Cuando finaliza la ejecución de una instrucción, se empieza la ejecución de la siguiente instrucción. En la figura 2 se muestra la ejecución de instrucciones en un procesador con sus etapas segmentadas, En cada ciclo de reloj una instrucción acaba una etapa y pasa a la siguiente. En la CPU hay varias instrucciones ejecutándose simultáneamente, cada una realizando una etapa. Podemos observar comparando ambas figuras que en lo que tarda en ejecutarse tres instrucciones en la CPU no segmenta, terminan su ejecución mas de siete instrucciones en la CPU segmentada, obteniendo una considerable ganancia de prestaciones.

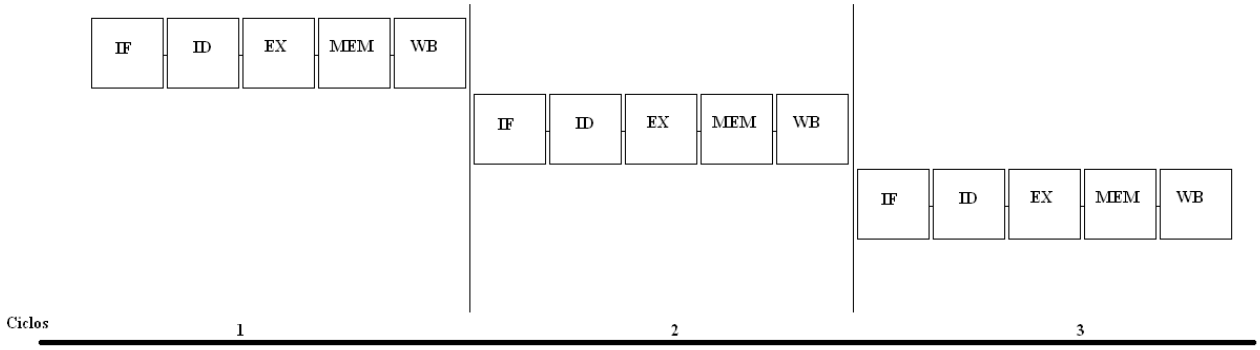

Figura 1: Ejecución de instrucciones en CPU no segmentada.

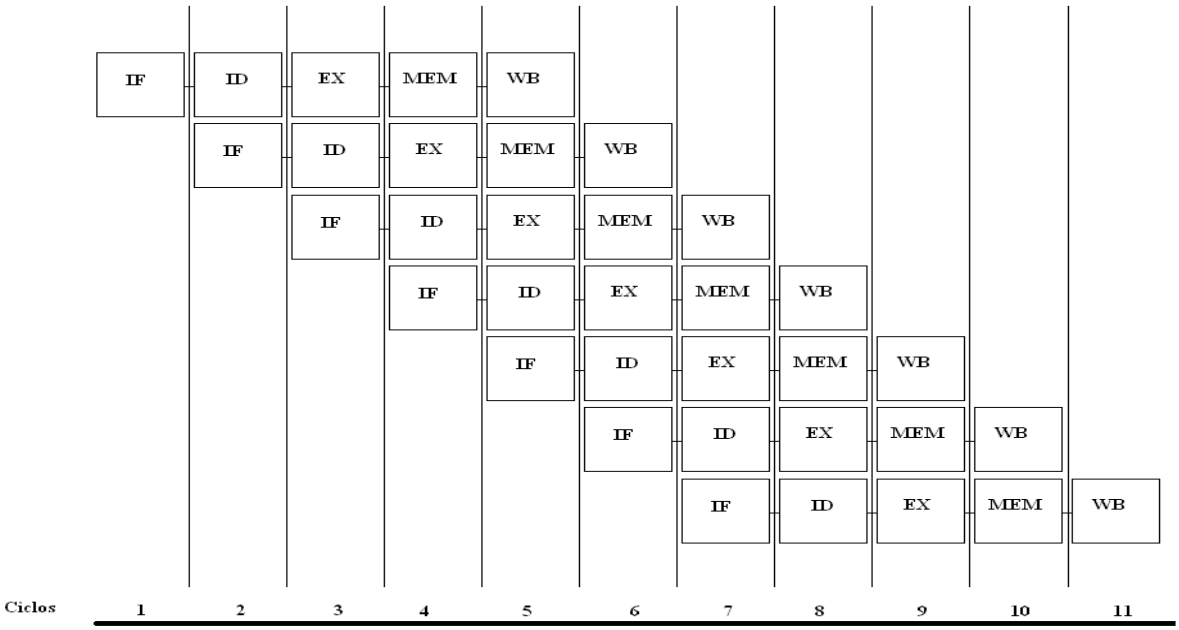

Figura 2: Ejecución de instrucciones en CPU segmentada.

No obstante en la ejecución de todo programa se producen situaciones que impiden que se ejecute la siguiente instrucción del flujo de instrucciones, lastrando las prestaciones de la CPU e impidiendola

alcanzar sus máximas prestaciones teóricas. A estas situaciones se les conoce como riesgos. Básicamente hay tres tipos de riegos:

1- Riesgos estructurales: Son aquellos que surgen de conflictos por los recursos. Es decir el hardware no puede soportar todas las ejecuciones solapadas de instrucciones simultáneamente por falta de recursos.

2- Riesgos de datos: Se producen cuando una instrucción necesita el resultado de otra instrucción previa que aun no ha acabado su ejecución. Los riesgos de datos pueden ser de tres tipos:

2.1- RAW : Una instrucción trata de leer un registro antes de que una instrucción anterior haya escrito sobre dicho registro.

2.2- WAR : Una instrucción trata de escribir en un registro antes de que una instrucción anterior haya leído dicho registro.

2.3- WAW : Una instrucción trata de escribir en un registro antes de que una instrucción haya escrito sobre dicho registro. Las escrituras se hacen en orden incorrecto dejando en el registro destino el valor de la instrucción ejecutada en primer lugar y no la ultima como debería ser.

3- Riesgos de control: Se producen por la ejecución de saltos condicionales y otras instrucciones que cambian el contador de programa, donde aun no se conoce por donde seguirá el flujo de programa hasta que dicha instrucción finalice.

Para impedir el bloqueo en el flujo de instrucciones debido a estos riesgos, que lastra de manera considerable las prestaciones de la CPU, surge la ejecución fuera de orden. La idea clave consiste en que la CPU pueda lanzar a ejecución instrucciones posteriores a la que se ha parado, alterando dinámicamente el flujo del programa. Evitando que una instrucción parada por algún tipo de riesgo, afecte a las instrucciones que le siguen.

De manera general, sin entrar en mucho detalle, podemos evitar paradas en el flujo ejecución de instrucciones en la CPU por riesgos de datos añadiendo una nueva etapa a la ruta de datos segmentada. Esta nueva etapa llamada Issue (emisión), se encarga de gestionar las instrucciones una vez decodificadas y enviarlas a ejecución cuando todos sus operandos necesarios están disponibles. Las instrucciones que aun tienen dependencias de datos y no pueden lanzarse a ejecución esperan en una cola de instrucciones hasta que se cumplen todos sus requisitos. Con estos cambios podemos evitar los bloqueos por riegos de datos ejecutando instrucciones independientes de manera desordenada. Para evitar el bloqueo por riesgos de control existe la ejecución especulativa.

La ejecución especulativa se basa en la predicción dinámica de saltos. Esta predicción se realiza con ayuda del predictor de saltos, esta estructura hardware se utiliza para predecir el comportamiento de las instrucciones de salto basándose en sus anteriores ejecuciones , y sirve para seleccionar de forma especulativa que instrucciones se deben ejecutar después de un salto condicional. Para garantizar el buen funcionamiento del sistema se debe confirmar, una vez la instrucción de salto ha terminado su ejecución, que la predicción era correcta y en caso contrario que las instrucciones lanzadas especulativamente no han variado el estado de la maquina. Para esto se añade una nueva etapa de confirmación llamada Commit. Esta nueva etapa se efectúa al final de la ruta de datos y sirve para confirmar la ejecución de la instrucción, por lo que solo se lleva a cabo si se confirma la ejecución de dicha instrucción. Las instrucciones aunque se ejecutan fuera de orden, realizan esta etapa en

orden. Cuando se realiza esta etapa es cuando se confirma que la instrucción debe ejecutarse y es entonces cuando se actualizan los registros, la memoria y se reconocen las excepciones.

La ruta de datos modificada resultante puede verse en la figura 3 Las instrucciones se lanzan y se decodifican (IF), una vez hecho esto en la etapa de emisión (Issue), se guardan en una cola de instrucciones especialmente diseñada llamada reorder buffer (ROB), donde se quedaran hasta que se confirme o se descarte su ejecución. Cuando sus operando están listos se ejecutan en su correspondiente unidad funcional y sus resultados son guardados en el ROB hasta que se ejecute la etapa de confirmación (Commit), entonces se actualizaran los registros o la memoria.

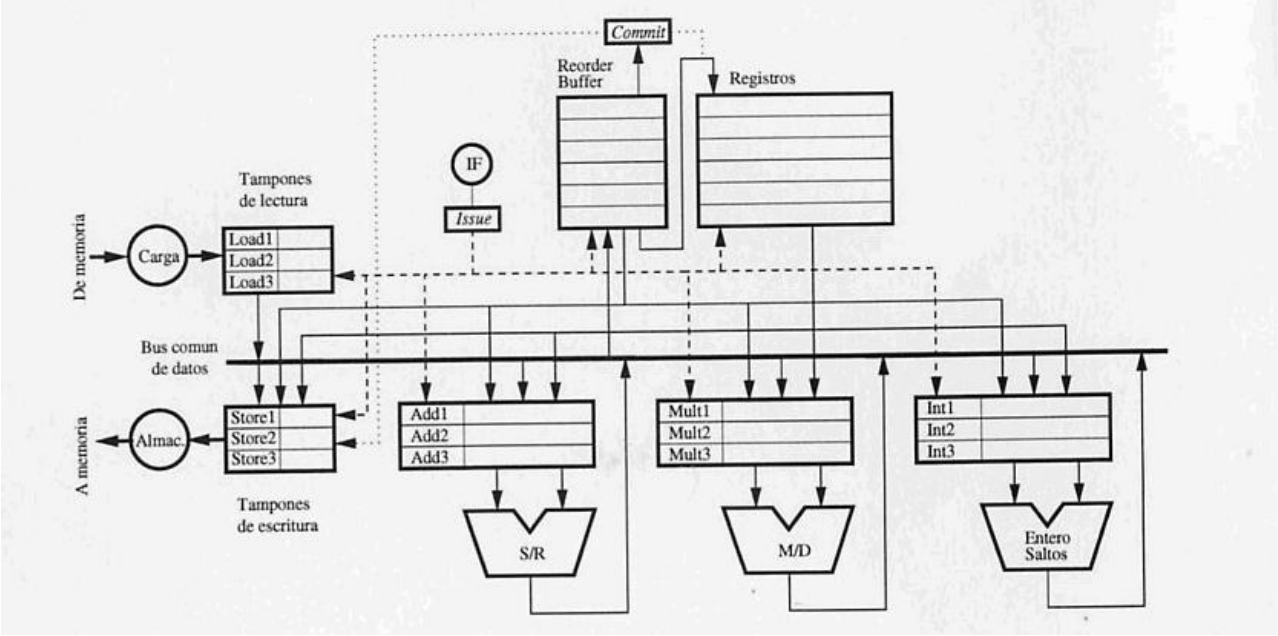

Figura 3: Ruta de datos modificada.

Esta técnica representa una mejora en las prestaciones de la CPU significativa, aunque requiere de una alta complejidad lógica para funcionar correctamente. Debido a esto los procesadores empotrados de mas bajo coste no utilizan esta técnica ya que se necesita una gran superficie de silicio para construir dicho hardware.

3.2.2 - EL SUBSISTEMA DE MEMORIA

La memoria es la encargada de guardar los datos e instrucciones que ejecuta la CPU. Para que una CPU no vea reducido su rendimiento debería poder recibir, sin que esto suponga una reducción en la frecuencia de la CPU , al menos, una instrucción por ciclo. Sin embargo este tipo de memorias es caro, ya que debe incluirse en el encapsulado del microprocesador ocupado una gran superficie de silicio. Para resolver esto de una manera económica, surge la idea de la jerarquía de memoria.

La jerarquía de memoria surge en respuesta a la necesidad de los programadores de requerir cantidades de memoria rápida ilimitadas. Se basa en el principio de localidad, los programas no acceden al código o los datos uniformemente sino que favorecen una parte determinada de su espacio de direcciones en cada momento. Este principio tiene dos vertientes. El principio de localidad espacial y el de localidad temporal. El principio de localidad temporal dice que si se referencia una posición de memoria, esta tendera a ser referenciado nuevamente pronto. Mientras

que el principio de localidad espacial dice que si se accede a una posición de memoria, se tendera a referenciar a las posiciones cercanas a esta en un lapso de tiempo corto.

Conocemos como jerarquía de memoria a la división de manera piramidal de la memoria del sistema. Sabiendo que la memoria mas rápida es también la mas cara y viceversa, podemos dividir la memoria del sistema en diversos niveles, de diferentes tamaños y velocidades, donde de manera ordenada los primeros niveles contendrán la memoria mas pequeña y rápida y los últimos la mas grande y lenta. De manera que todos los datos de un nivel inferior se encuentran contenidos en los niveles superiores, conteniendo la memoria mas rápida los objetos de memoria que mas probabilidades, según el principio de localidad, tienen de ser accedidos próximamente. En la siguiente figura, la figura 4, se puede apreciar la típica organización piramidal del subsistema de memoria.

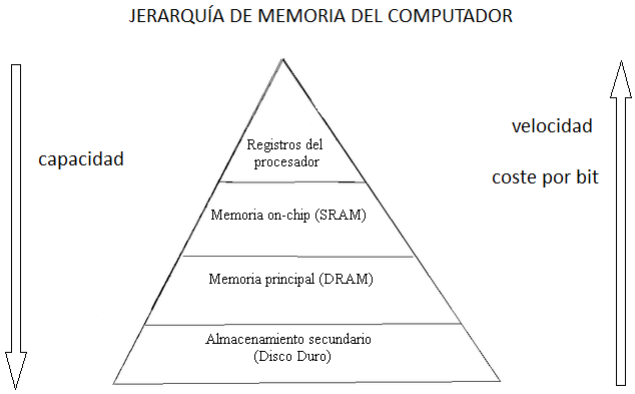

Figura 4: Jerarquía de memoria.

A continuación explicaremos cada uno de los niveles de la jerarquía, poniendo especial interés en las memorias on-chip. Los registros del procesador son pequeñas memorias que se encuentran junto a la CPU y que se encargan de guardar datos y direcciones con las que operan las instrucciones. Casi todas las instrucciones ejecutadas en la CPU involucran de una forma u otra a los datos de uno o varios registros, por lo que para que no supongan un grave cuello de botella en el sistema deben funcionar a la misma velocidad que la CPU. La capacidad de estos registros es muy limitada, cada registro de la CPU es capaz de guardar un solo dato o dirección de memoria.

En el siguiente nivel de la jerarquía se encuentran las memorias on-chip. En esta memoria trataremos sobre distintas configuraciones de memorias on-chip en sistemas empotrados, por lo que pondremos especial énfasis en explicar este tipo de memorias. Suelen ser memorias estáticas de acceso aleatorio (SRAM) ya que en comparación con las memorias dinámicas de acceso aleatorio (DRAM) son más rápidas y presentan un menor consumo, aunque tienen un mayor coste económico. Este tipo de memorias se les conoce como memorias on-chip por que suelen estar fabricadas compartiendo área de silicio con la CPU, en el propio chip, por lo que suelen ser de pequeño tamaño en los procesador empotrados mas sensibles al coste. Además son las culpables de la mayor parte del consumo energético gastado por el procesador.

Este tipo de memorias presentan un tiempo de acceso muy bajo, funcionando a la misma velocidad que el procesador, permitiendo salvar la diferencia de velocidad entre la CPU y la memoria principal. Su buen funcionamiento se basa en los principios de localidad, guardando en ellas, en

cada momento, las instrucciones y datos que mas posibilidades tienen de ser accedidos en los próximos ciclos de reloj. En esta memoria trataremos los dos principales tipos de memorias on-chip, la memoria caches y la SPM.

La memoria cache es el tipo de memoria on-chip mas utilizado ya que gracias a su funcionamiento autónomo, basado en los principios de localidad, le permiten adaptarse a cualquier escenario de ejecución, ofreciendo siempre un gran rendimiento. Toda memoria cache esta divida en bloques de memoria llamados lineas de cache, cada una de las lineas esta formada por una o mas instrucciones u operandos, habitualmente 4 u 8 y el tamaño de esta puede variar según la memoria cache, pudiendo variar de unos pocos Kilobytes hasta el Megabyte. Una linea de cache es la unidad básica de transferencia, cada movimiento desde la memoria principal cubrirá siempre una línea de cache. Además cada línea lleva asociada una etiqueta con la dirección de memoria de ese bloque, que la identifica. Cada vez que la CPU quiera acceder a una dirección de memoria, la memoria cache comparará dicha dirección con las etiquetas de las lineas de cache guardadas en esta, en caso de acierto suministrará a la CPU la instrucción o dato requerido mientras que en caso de fallo elegirá un bloque como víctima y lo sustituirá con el bloque que contenga la dirección de memoria requerida, obteniendo dicho bloque mediante una transferencia desde la memoria principal.

Cada memoria cache posee una serie de características, a tener en cuenta en la fase de diseño del sistema, que diferencian unas de otras. Según donde puede ubicarse un bloque de memoria dentro de la cache pueden dividirse en memorias caches de correspondencia directa, totalmente asociativas o asociativas por conjuntos. Las memorias caches de correspondencia directa son aquellas en que cada bloque solo puede ir ubicado en una línea de cache. Por contra una cache totalmente asociativa es aquella en la que un bloque puede ubicarse en cualquiera de sus lineas de cache. Mientras que una cache asociativa por conjuntos es aquella donde un bloque de memoria puede ubicarse en un conjunto restringido de líneas de cache, los conjuntos están formados por varias lineas de cache, un bloque primero sera asignado a un conjunto y después podrá ubicarse en cualquiera de las líneas que forman dicho conjunto. Si hay n líneas de cache en un conjunto se dice que la cache es una cache asociativa por conjuntos de n vías. La principal ventaja de una cache de correspondencia directa es su sencillez, para saber si un bloque de memoria esta en la cache solo hace falta comparar una etiqueta, lo que hace que este tipo de caches necesiten una lógica de control mas sencilla y sean mas veloces.

En la figura 5 vemos un ejemplo. La figura muestra las tres configuraciones de cache antes comentadas, y refleja la cantidad de etiquetas que deben compararse cada vez que se busca un bloque de memoria en la cache. La figura muestra un escenario donde se busca el bloque de memoria numero 7 en una hipotética cache de 8 lineas. Se puede observar como en la cache de correspondencia directa dado que ese bloque solo puede ubicarse en la posición solo deberá compararse una etiqueta. Al contrario en la cache totalmente asociativa, ese bloque podría ubicarse en cualquier posición de la cache por lo que deben compararse todas las etiquetas, mientras que en la cache asociativa de dos vías deberán hacerse dos comparaciones ya que ese bloque puede encontrase en cualquiera de las posiciones que forman el conjunto 3. El numero de comparaciones repercute activamente en el consumo en la velocidad de la cache en caso de acierto y la complejidad de la lógica de control que necesita la cache. Actualmente la mayoría de procesadores actuales utiliza cache de correspondencia directa o asociativas de hasta 4 vías, reservándose niveles de asociatividad superiores solo para memoria de un tamaño muy pequeño, del orden de centenares de bytes.

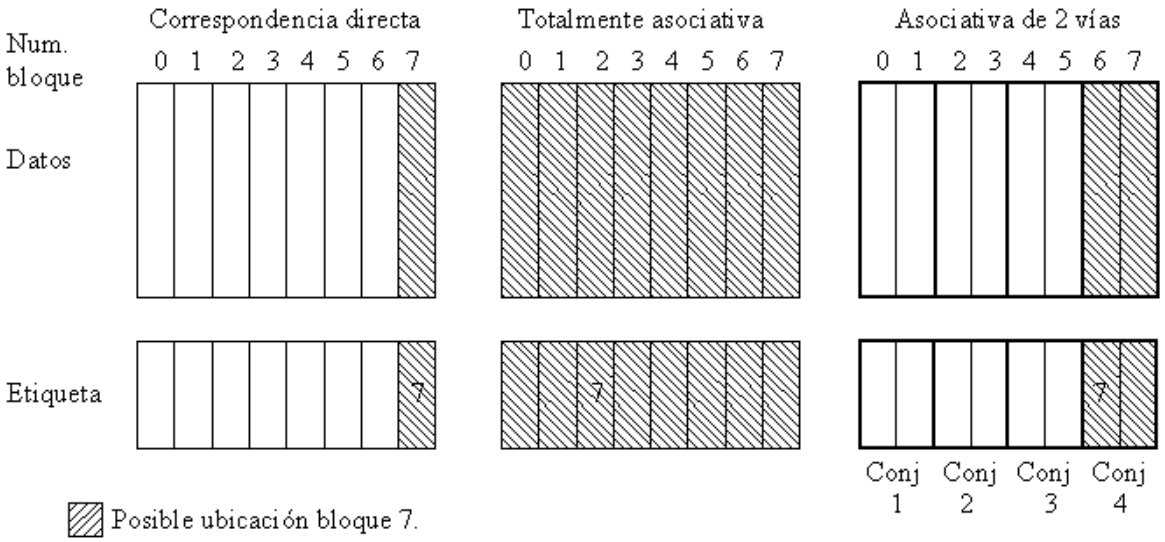

Figura 5 Comparación de distintos tipos de memorias caches.

Otro aspecto a tener en cuenta es el tamaño de las etiquetas. En la figura 6 vemos como se divide una dirección de memoria en tres campos para encontrar un dato o instrucción en la memoria cache. Los primeros d bits de la dirección son usados para el desplazamiento dentro de un bloque de memoria, los bits utilizados dependen del tamaño de línea que tenga la memoria cache. Los siguientes i bits son utilizados para saber en que conjunto dentro de la memoria cache puede estar dicho bloque. En una memoria totalmente asociativa i seria igual a cero, ya que el bloque podría estar en cualquier línea de cache, mientras que en una memoria de correspondencia directa se usaran la mayor cantidad posible de bits para saber donde puede ubicarse dicho bloque ya que dicho bloque solo podrá estar en una posición de la memoria cache. Los últimos e bits se utilizan como etiqueta, son los bits que se deben comparar para saber si un bloque esta o no en la memoria cache. Estos bits deben guardarse en la memoria cache por lo que el tamaño de la etiqueta repercutirá activamente en el tamaño físico, en la oblea de silicio , y consumo de la memoria cache, por lo tanto cuanto más pequeña sea la etiqueta mejor.

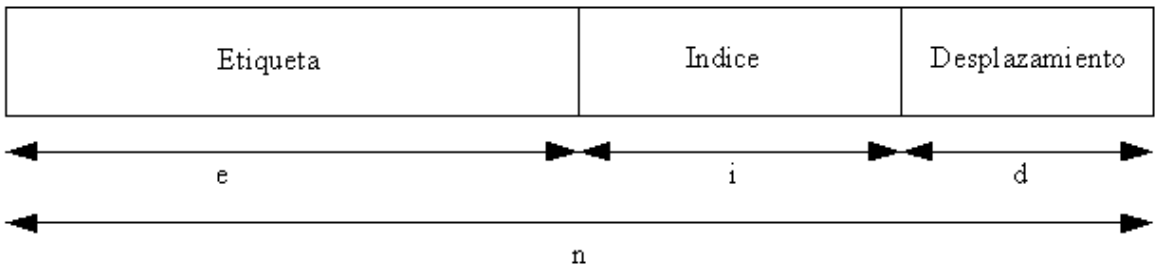

Figura 6: División de una dirección de memoria.

Otra característica importante que diferencia entre unas caches y otras es la elección del bloque a remplazar cuando se produce un fallo. En caches de correspondencia directa simplemente no hay elección, ya que el bloque solo puede ir ubicado en un lugar simplemente sustituirá al bloque que estaba ocupando ese lugar, esto simplifica en buena medida la lógica de control de la cache.

Mientras que en caches asociativas se utilizan distintas estrategias para elegir el bloque victima, las más comunes son la elección aleatoria y la elección del bloque menos recientemente usado (Least Recently Used LRU), no obstante existen otras estrategias como el uso del algoritmo First In First Out (FIFO). La elección aleatoria consiste en como su propio nombre indica elegir el bloque a sustituir de manera aleatoria entre todos los posibles, siendo una estrategia simple y sencilla de construir en el hardware. La elección del bloque victima mediante el algoritmo LRU consiste en seleccionar como bloque a sustituir el bloque que hace mas tiempo que no ha usado la cache. Aunque este algoritmo presenta unos buenos resultados teóricos, es caro de implementar en la realidad, requiere una gran lógica de control para llevarlo a cabo lo que hace que en la práctica en cache con muchas vías o totalmente asociativas sea inviable. Por ultimo el algoritmo FIFO simplemente selecciona como bloque a sustituir a aquel que lleva mas tiempo en la cache, es decir al bloque que entro antes en la cache.

La última característica importante que puede variar de una cache a otra es su política de coherencia en caso de escritura. Las escrituras solo suceden cuando tenemos datos en la cache, en caso de que se produzca un cambio en dichos datos, deben ser actualizados en los niveles inferiores de la jerarquía de memoria. Hay dos formas de hacer esta actualización mediante escritura directa (write-through), los datos son copiados en el mismo momento en que se producen los cambios a memoria principal, o mediante postescritura (write-back), donde los datos son copiados a memoria principal solo cuando la linea de cache va a ser remplazada. Ambos métodos tienen sus ventajas e inconvenientes, en el caso de la escritura directa siempre se mantiene la coherencia entre la memoria principal y la memoria cache, pero a costa de aumentar el tiempo necesario para realizar una escritura, mientras que en la postescritura se utiliza menos ancho de banda, las transferencias entre memoria cache y memoria principal son menos frecuentes, solo se accede a memoria principal en caso de que sea imprescindible, pero requiere aumentar la complejidad de la memoria cache, ya que se hace necesario tener controlados los bloques que han sido modificados.

Es común que en un mismo procesador habiten varios tipos de memorias caches divididas por niveles o por el tipo de contenidos que guarden. Las memorias cache pueden usarse para guardar solo código ejecutable, usarse para guardar solo datos o usarse de forma mixta para guardar en una misma memoria tanto datos como instrucciones. Además pueden estructurarse en distintos niveles, los procesadores mas avanzados como el Itanium tienen actualmente hasta cuatro niveles de cache, estando incluso el ultimo nivel fuera del propio chip.

Todas las características comentadas anteriormente, junto a la carga ejecutada influyen de manera activa en el rendimiento de la cache, es decir determinan la tasa de fallos y aciertos que se producen en la memoria cache. Los fallos que se producen en una memoria cache son debidos a una de estas tres circunstancias:

- Fallos forzosos: La primera vez que se solicita un bloque que no esta en cache deberá ser transferido de memoria principal a la memoria cache obligatoriamente.
- Fallos por capacidad: Se producen cuando la cache, debido a su falta de capacidad, no puede contener todos los bloques necesarios durante la ejecución de un programa, de manera que ciertos bloques se descartaran y se recuperaran posteriormente.
- Fallos por conflicto: En caches asociativas por conjuntos o de correspondencia directa se producen estos fallos cuando varios bloques de memoria compiten por un mismo conjunto de líneas, lo que requerirá descartar bloques que posteriormente volverán a ser referenciados.

El objetivo final de toda memoria cache es encontrar el tipo de memoria cache que obtenga las mejores prestaciones al coste mas bajo posible, sobretodo cuando hablamos de procesadores para sistemas empotrados. Las prestaciones de una cache se miden en la cantidad de aciertos obtenidos por el numero de accesos a la memoria. Se produce un acierto cuando la CPU referencia una dirección de memoria que se encuentra en la cache, la cache suministrara el contenido de dicha dirección a la CPU sin ninguna demora. Sin embargo si dicha dirección no se encuentra en la cache, la CPU deberá esperar a que la memoria principal transfiera el bloque de memoria a la cache para seguir funcionando. La velocidad del sistema depende en buena medida de una alta tasa de aciertos en la cache.

Como hemos visto la cache funciona de manera autónoma, sin que el programador deba preocuparse de los contenidos que esta guarda. Sin embargo aunque esto le haga adaptarse de manera correcta a cualquier carga de trabajo ejecutada, su poco determinismo presenta problemas cuando se trata de ejecutar programas en sistemas de tiempo real. La difícil tarea de saber los contenidos en cada momento que contiene la memoria cache hace que calcular WCET sea una tarea complicada en este tipo de sistemas. Es por esto que en muchos de estos sistemas se opte por usar otro tipo de memorias on-chip mas deterministas denominadas scratchpad memories (SPM).

La SPM es un tipo de memoria rápida SRAM mas pequeña y rápida que la memoria cache. Se suele utilizar para guardar los datos e instrucciones mas accedidos por la CPU. A diferencia de la memoria cache, la SPM contiene direcciones de memoria mapeadas dentro del rango de direcciones del sistema, por lo que cualquier acceso a estas direcciones devolverá el dato o instrucción que esta contenga sin demora para la CPU. De esta manera se prescinde de las etiquetas , lo que repercute en que esta memoria ocupe menos espacio en la oblea de silicio que las memorias cache de tamaño semejante, además podemos prescindir de toda la lógica de control usada para la comparación de etiquetas y de las comparaciones usadas para saber si la dirección de memoria requerida se encuentra en la cache, lo que nos da como resultado una memoria más sencilla, más rápida, más barata y con un menor consumo.

Sin embargo al carecer de toda la lógica de control de la memoria cache, se pierde el comportamiento autónomo de esta memoria, los datos e instrucciones deben ser introducidos en ella por el programador o por el compilador. Este factor hace que este tipo de memorias sean poco practicas en sistemas de propósito general sin embargo se presentan como una buena opción en sistemas empotrados donde conocemos la carga de trabajo del sistema ya que conociendo la carga de trabajo que tiene el sistema podemos realizar un estudio sobre esta para establecer que instrucciones o datos debemos meter en la SPM y en que momento. Además su total predictibilidad hace más sencillo el calculo del WCET en sistemas de tiempo real.

En la figura 7 vemos como quedaría el mapa de memoria en un sistema con una SPM. Como podemos observar la memoria SPM se encuentra mapeada dentro del conjunto de direcciones de memoria accesibles. Cuando la CPU pida una dirección que se encuentre en las primeras P-1 direcciones de memoria sera la SPM la encargada de suministrar la instrucciones sin que esto supongo ninguna demora. Mientras que si la CPU referencia una posición de memoria superior a P-1 sera la memoria principal la encarga de servirle el objeto de memoria con el retardo correspondiente.

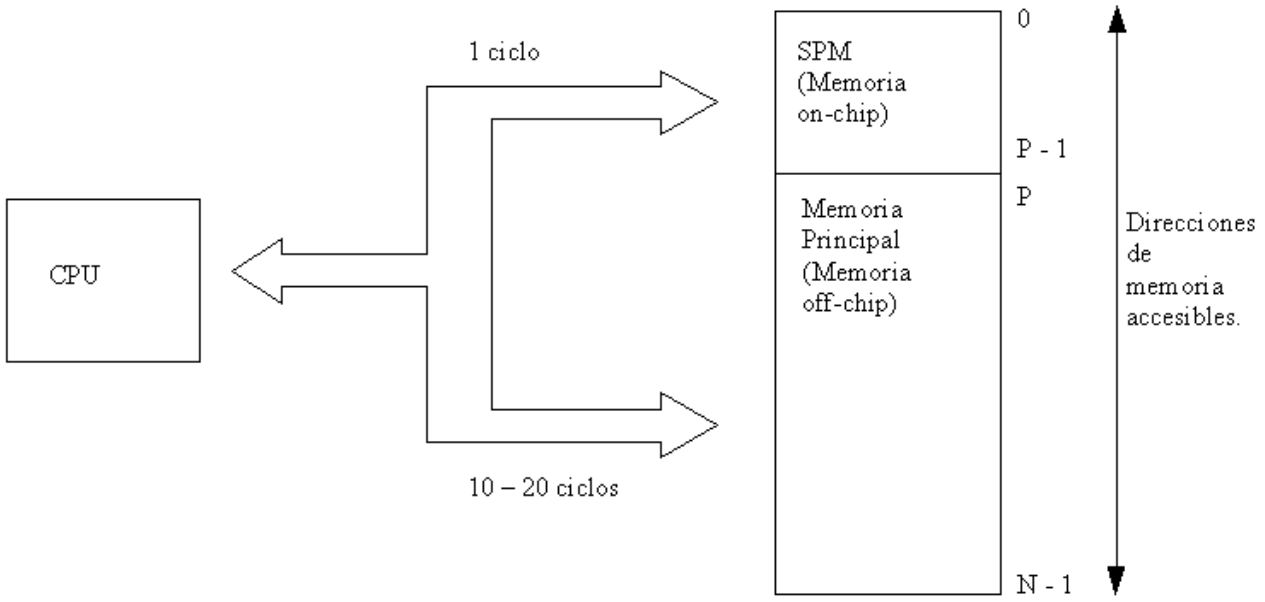

Figura.7: Sistema con SPM.

Para transferir datos e instrucciones de la memoria principal a la SPM tenemos dos opciones. A menudo para la transferencia de instrucciones y datos a la SPM se utiliza el acceso directo a memoria (Direct Memory Access, DMA). Mediante la DMA se permite el acceso a la SPM a memoria principal sin que la CPU se vea afectada. Como esta operación no ocupa a la CPU, esta puede seguir ejecutando instrucciones sin quedarse bloqueada. La segunda opción es que la CPU se encargue de las transferencias utilizando instrucciones de load y store para tal fin. Esta segunda opción es la que hemos considerado en esta memoria debido a que el coste monetario de dicha opción es mas bajo.

Como hemos visto ambas memorias on-chip tienen sus ventajas e inconvenientes y se adecuan mas a ciertas situaciones. Aunque hayamos visto ambas memorias on-chip como memorias contrapuestas lo cierto es que es común encontrarse con sistemas que posean ambos tipos de memorias, aprovechando así las particularidades de ambas.

Bajando por la escala de la jerarquía de memoria, el siguiente nivel que nos encontramos es la memoria principal. La memoria principal es una memoria dinámica de acceso aleatorio (DRAM). Podemos distinguir entre dos tipos, la memoria de acceso aleatorio (Random Access Memory, RAM) que es encargada de contener los datos e instrucciones de las tareas que se encuentran en ejecución de forma temporal y la memoria de solo lectura (Read Only Memory, ROM) que suele utilizarse en los sistemas empotrados mas sencillos sustituyendo al almacenamiento secundario, para guardar el software y los datos fijos y constantes. Mientras que el contenido de la memoria RAM es temporal y se pierde cuando se interrumpe la corriente eléctrica, la memoria ROM es invariable y no volátil es decir siempre mantiene la información. Este tipo de memorias son de un tamaño sensiblemente mayor a las memorias on-chip aunque también son mas lentas, un acceso a estas memorias puede suponer una espera de unos 10 a 20 ciclos. Estas memorias se encuentran fuera del microprocesador entre la memoria on-chip y la memoria secundaria y se comunica con la memoria on-chip o la CPU mediante el bus principal del sistema.

Por último en el nivel mas bajo de la jerarquía nos encontramos con los dispositivos de

almacenamiento secundario como podría ser un disco duro, una disquetera, un lector de tarjetas, ect... Este tipo de memoria se caracteriza por tener una gran capacidad, ser independientes de la memoria principal y la CPU y por mantener la información en caso de perdida del suministro eléctrico.

3.2.3 - SISTEMA DE ENTRADA / SALIDA

Los sistemas de entrada y salida engloban a todos los periféricos que son utilizados por el computador para enviar o recibir información desde o hacia el exterior. El subsistema de entrada acepta datos del exterior para ser procesados mientras que el subsistema de salida transfiere los resultados hacia el exterior. Estos dispositivos suelen tener un tiempo de operación mucho mayor que el procesador, a la hora de calcular los tiempos de ejecución de las tareas deberá tenerse en cuenta estos dispositivos.

CAPÍTULO 4 : HERRAMIENTAS UTILIZADAS

En este capitulo hablaremos sobre las distintas herramientas que hemos utilizado para simular los distintos tipos de memorias on-chip. Explicaremos tanto el simulador utilizado como los benchmarks usados así como el montaje experimental.

4.1 - VATIOS: SIMULADOR DE PROCESADOR CON ESTIMACION DE POTENCIA.

En este proyecto para simular las distintas arquitecturas de memorias on-chip hemos utilizado el simulador Vatios [14]. Vatios es un simulador basado en el popular simulador SimpleScalar [15] que de manera semejante al simulador Wattch [16] añade un modelo para calcular el consumo energético de las entidades simuladas.

4.1.1 - SIMPLESCALAR

Simplescalar[15] es un simulador de arquitecturas de computadores escrito en C. Se utiliza para poder verificar comparaciones de rendimiento entre varias máquinas diferentes sin construir físicamente ninguna de estas máquinas. Goza de una gran popularidad, es de libre distribución para ámbitos académicos y es actualmente el simulador mas utilizado por los investigadores para evaluar los diseños de sus publicaciones.

Simplescalar fue creado por Tod Austin en 1992 como parte de un proyecto en la universidad de Wisconsin aunque las primeras versiones incluyeron contribuciones de Doug Burger y Guri Sohi. Hoy, Simplescalar es desarollado y apoyado por Simplescalar LLC.

El objetivo de todo este tipo de simuladores es acelerar el desarrollo del hardware, empleando modelos software de lo que se desea construir. Estos simuladores se utilizan para desarrollar, probar y comparar distintas alternativas de diseños de hardware sin tener que construir físicamente ningunos de los diseños. Los distintos diseños se simulan con la carga de trabajo elegida y a continuación se valora la corrección del diseño y su rendimiento.

Simplescalar contiene un conjunto de simuladores y herramientas que ayudan a validar y evaluar cualquier diseño. El conjunto de simuladores que se ofrecen con Simplescalar incluyen, dos procesadores funcionales, sim-fast y sim-safe, de funcionamiento poco detallado, no tienen en cuenta el comportamiento de la memoria cache o de la ruta de datos segmentada, no modelan la temporización ni producen ninguna salida especial. Un simulador especial llamado sim-profile que proporciona gran cantidad de información estadística sobre los programas ejecutados. Dos simuladores del sistema de memoria, sim-cache y sim-cheetah, capaces de simular sistemas con multiples niveles de memorias cache de instrucciones o datos, en el cual cada uno puede ser configurado con distinto tamaño u organización. Un simulador del predictor de saltos llamado sim-bpred que puede simular diferentes esquemas de predictores de saltos. Y por ultimo un simulador mas complejo llamado sim-outorder capaz de simular un procesador superescalar fuera de orden. Este procesador modela la temporización interna de un procesador ciclo a ciclo. Todo los simuladores comparten partes del mismo código. El simulador utilizado en este proyecto se basa en el simulador sim-outorder. En la figura 8 podemos ver el conjunto de simuladores que forman parte de Simplescalar, del más sencillo al más complejo. Además Simplescalar también incluye herramientas para depurar y hacer trazas de los programas que ejecutamos en el simulador.

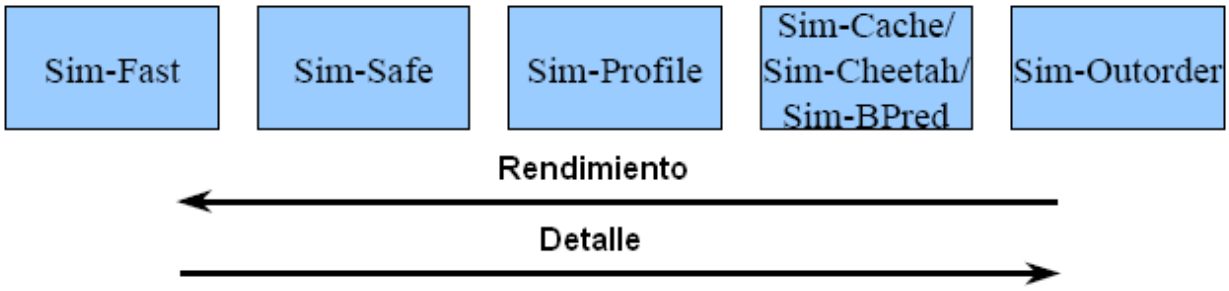

Figura 8: Simuladores que contiene Simplescalar.

Tal como hemos comentado en este proyecto hemos utilizado un simulador basado en el simulador sim-outorder que añade resultados sobre el consumo energético de las entidades que forman el simulador. Este simulador esta basado en la arquitectura MIPS IV ISA con unos pequeños cambios y añadidos. El conjunto de instrucciones que soporta el simulador se muestran en la figura 9 .

Control
j - jump
jal - jump and link
jr - jump register
jalr - jump and link register
beq - branch == 0
bne - branch != 0
blez - branch <= 0
bgtz - branch > 0
bltz - branch < 0
bgez - branch >= 0
bct - branch FCC TRUE
bcf - branch FCC FALSE

Load/Store
lb - load byte
lbu - load byte unsigned
lh - load half (short)
lhu - load half (short) unsigned
lw - load word
dlw - load double word
l.s - load single-precision FP
l.d - load double-precision FP
sb - store byte
sbu - store byte unsigned
sh - store half (short)
shu - store half (short) unsigned
sw - store word
dsw - store double word
s.s - store single-precision FP
s.d - store double-precision FP

addressing modes:
(C)
(reg+C) (with pre/post inc/dec)
(reg+reg) (with pre/post inc/dec)

Integer Arithmetic
add - integer add
addu - integer add unsigned
sub - integer subtract
subu - integer subtract unsigned
mult - integer multiply
multu - integer multiply unsigned
div - integer divide
divu - integer divide unsigned
and - logical AND
or - logical OR
xor - logical XOR
nor - logical NOR
sll - shift left logical
srl - shift right logical
sra - shift right arithmetic
slt - set less than
sltu - set less than unsigned

Floating Point Arithmetic
add.s - single-precision (SP) add
add.d - double-precision (DP) add
sub.s - SP subtract
sub.d - DP subtract
mult.s - SP multiply
mult.d - DP multiply
div.s - SP divide
div.d - DP divide
abs.s - SP absolute value
abs.d - DP absolute value
neg.s - SP negation
neg.d - DP negation
sqrt.s - SP square root
sqrt.d - DP square root
cvt - int., single, double conversion
c.s - SP compare
c.d - DP compare

Miscellaneous
nop - no operation
syscall - system call
break - declare program error

Figura 9: Instrucciones soportadas por Simplescalar.

Las instrucciones del Simplescalar son todas de 64 bites y están divididas en tres formatos, de registro o tipo r, inmediato o tipo i y salto o tipo j. Las instrucciones de tipo registro son usadas para instrucciones computaciones entre registros. Las de tipo inmediato soportan la inclusión de una constante de 16 bits y por ultimo las de tipo salto disponen de 24 bits para especificar la dirección destino del salto. En la figura 10 vemos la arquitectura de los distintos tipos de instrucciones.

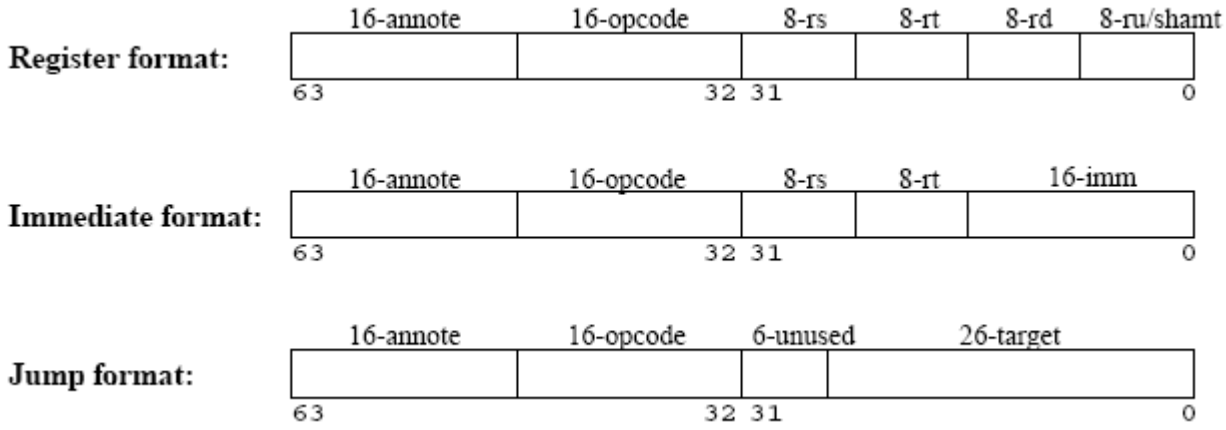

Figura 10 : Arquitectura de los distintos formatos de instrucción del Simplescalar.

El simulador sim-outorder simula un procesador con ejecución fuera de orden con ejecución especulativa altamente configurable. La arquitectura de la ruta de datos segmentada del simulador puede verse en la figura 11 . Simula una ruta de datos segmentada de seis etapas, búsqueda, decodificación, emisión, ejecución, escritura y commit. Como estructuras especiales destacar que el simulador dispone de una cola de instrucciones donde se alojan las instrucciones entre la etapa de búsqueda y decodificación, además las instrucciones una vez descodificadas son enviadas a una estructura especial conocida como register update unit (RUU), que funciona como ROB, donde las instrucciones esperaran hasta que se confirme su ejecución.

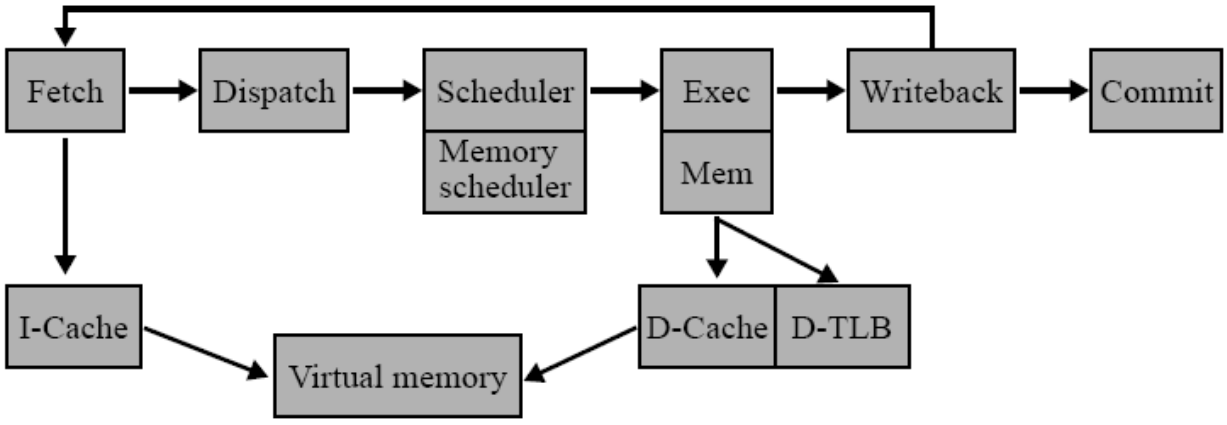

Figura 11: Ruta de datos segmentada del simulador sim-outorder.

Tal como hemos comentado el simulador es altamente configurable, las opciones que incluye el simulador para simular la arquitectura del procesador deseada se muestran en el anexo A.

4.1.2 - WATTCH

Wattch [16] es un simulador basado en Simplescalar desarrollado en el año 2000 por David Brooks en la universidad de Princeton. Este simulador añade un modelo de consumo y potencia al

simulador Simplescalar. Wattch toma como entrada la definición del simulador y una traza o ejecutable a simular y tras su simulación muestra una serie de datos sobre el consumo energético de cada unidad funcional del procesador simulado. Este simulador cuenta con una gran popularidad y ha servido para validar una gran cantidad artículos de investigación sobre consumo energético en los últimos años.

Wattch muestra cuatro valores de consumo como resultado para cada unidad del procesador. Muestra un consumo para cada unidad aplicando cuatro tipos de clock gating, el clock gating es una técnica usada para ahorrar energía usada en muchos circuitos síncronos que consiste en inhibir la señal de reloj a zonas inactivas del sistema, evitando la conmutación en esos ciclo y ahorrando energía.

Para calcular el consumo de cada unidad Wattch divide las unidades en cuatro tipos. Las tipo RAM y tipo CAM (memoria de contenido direccionable), unidades basadas en lógica computacional y distribución de reloj. Para cada uno de estos tipos calcula el consumo energético de una manera. En algunos casos, en el cálculo del consumo producido por memorias on-chip , se utiliza llamadas a funciones a la librería CACTI 1.0 [17].

CACTI es una herramienta para modelar memorias on-chip que incluye un modelo para calcular el área en silicio y el consumo energético de la memoria, esta escrito en C y puede ser utilizado libremente. La primera versión fue desarrollada en 1994 en el DEC Western Research Laboratory en Palo Alto, California. Actualmente esta desarrollado por HP Laboratories dentro de la compañía HP y su ultima versión es la 5.3 y puede accederse a ella via web http://quid.hpl.hp.com:9081/cacti/ [22].

4.1.3 - VATIOS

Vatios[14] es un simulador desarrollado en el año 2007 por Jose Antonio Victorio en la universidad de Zaragoza como proyecto final de carrera. Este simulador es de libre distribución y esta basado en Wattch[16] y en Simplescalar[15]. Este simulador simula la misma arquitectura que Simplescalar[15] y tiene la misma funcionalidad que Wattch[16] solo que le añade una serie de actualizaciones y ventajas.

Este simulador añade una serie de opciones para mejorar la funcionalidad del simulador Wattch. Opciones como la posibilidad de elegir la escala tecnológica y la frecuencia del procesador cuando se lanza a ejecución la simulación. Además utiliza una versión más actualizada de la herramienta CACTI[17] , la versión 4.2, que permite trabajar a distintas escalas tecnológicas sin tener que recompilar la herramienta. Esta actualización de CACTI[17] mejora los resultados del cálculo del consumo energético en el sub-banking de las memorias cache, siendo ahora más precisos.

Además a diferencia de Wattch[16], Vatios[15] no calcula la energía ciclo a ciclo, sino que guarda información sobre la actividad y uso de las distintas unidades para al final de la simulación calcular la energía consumida por el procesador. Lo que permite guardar la información sobre la actividad y uso de las unidades en un fichero y utilizarse en otra herramienta que suministra Vatios, llamada power-vatios ,que tiene un tiempo de ejecución despreciable, para hacer simulaciones variando modelos de potencia, escala tecnológica, frecuencia de funcionamiento y en general cualquier parámetro que no altere la temporización del procesador, ahorrando tiempo de tener que simular varias veces el mismo programa.

4.2 - BENCHMARKS

Todos los benchmarks utilizados en esta memoria para validar los distintos experimentos de memorias on-chip han sido obtenidos de The Mälardalen WCET research group [18]. Este grupo de trabajo mantiene un gran numero de benchmarks especialmente utilizados para evaluar distintas herramientas de trabajo sobre el WCET. Son benchmarks recolectados de distintos grupos de trabajos alrededor del mundo y especialmente preparados para que el calculo del WCET sea sencillo de realizar. Estos benchmarks han sido, en algunos casos, cuidadosamente modificados por nosotros para incluir las instrucciones precisas para la correcta utilización de cada una de las memorias on-chip, en el caso de que fuera necesario.

Los benchmarks y las características de cada uno de los programas que se incluyen los podemos encontrar en la web: http://www.mrtc.mdh.se/projects/wcet/benchmarks.html [20].

4.3 - SETUP EXPERIMENTAL

El flujo de trabajo cada uno de los experimentos, realizados en esta memoria, se muestra en la figura 12 .Empezamos con el programa escrito en código C, a continuación modificamos el program para añadir las instrucciones necesarias para la correcta utilización de la scratchpad, en el caso de que sea necesario. Compilamos el código con el compilador sslittle-gcc para obtener el código maquina. Este código maquina generado es simulado utilizando el simulador sim-vatios con el que obtendremos una traza de la ejecución que utilizara la herramienta power-vatios para calcular el consumo energético. En el caso del bloqueo de cache sera necesario crear un fichero con las direcciones de cache que introduciremos en esta, como explicaremos en el siguiente capítulo y que le introduciremos como parámetro en la simulación.

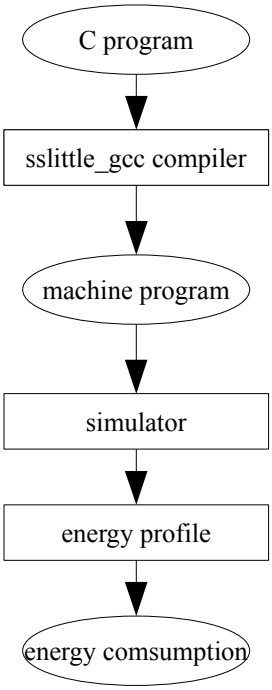

Figura 12 : Flujo experimental.

CAPÍTULO 5 : MEMORIA CACHE CON BLOQUEO

El bloqueo de contenidos en cache es una técnica hardware utilizada en sistemas empotrados para garantizar un comportamiento predecible del código ejecutado. Esta técnica consiste en fijar los contenidos de la memoria cache durante parte de la ejecución del programa, estos contenidos permanecerán invariables en la memoria cache, no se remplazaran por otros contenidos, mientras esta permanezca bloqueada. Fijar el contenido de la memoria cache es una técnica que no presenta excesiva dificultad, ya que muchos de los procesadores actuales dedicados a sistemas empotrados están diseñados para dar soporte a dicha técnica.

Aunque esta técnica pueda provocar una degradación de las prestaciones medias del procesador, el código ejecutado se vuelve más predecible lo que nos permitirá en sistemas de tiempo real, análisis sencillos, rápidos y practicables tanto del WCET como de la planificabilidad del sistema. Esta mejora del determinismo del sistema nos permitirá obtener cotas mas ajustadas en el cálculo del WCET, lo que repercutirá en una mejor planificabilidad del sistema utilizando los recursos de manera mas eficiente y hasta incluso en poder planificar sistemas que anteriormente no era planificables.

Existen dos posibles variantes para la fijación y bloqueo de contenidos en la memoria cache, la utilización del bloqueo de los contenidos de la memoria cache de manera estática y la utilización de manera dinámica. El uso del bloqueo de los contenidos de la memoria cache de manera estática consiste en fijar los contenidos durante toda la ejecución. Es la técnica mas sencilla, los contenidos son cargados en la memoria cache y una vez cargados permanecerán invariables durante toda la ejecución del sistema, es decir permanecerán estáticos, por lo tanto una vez fijado el contenido sera fácil decidir si un acceso a memoria sera un acierto o un fallo en la cache, lo que nos permitirá, de una manera menos compleja, obtener una cota del WCET mas ajustada. La otra opción, la utilización dinámica de las memorias caches con bloqueo, aunque es mas compleja, permitirá obtener unas mejores prestaciones del sistema sin perder el determinismo obtenido de la utilización estática. Consiste en permitir que los contenidos de la memoria cache se modifiquen, adaptándose a la ejecución de las tareas de manera semejante a una cache convencional, solo que la modificación de contenidos y los contenidos a introducir en la cache se realizan únicamente en ciertos instantes puntuales decididos durante la fase de diseño, mediante un análisis exhaustivo del sistema. Cuantos mas reemplazos se realicen mejores prestaciones tendrá el sistema, sin embargo más difícil sera determinar el comportamiento del sistema y ajustar la cota del WCET.

Como hemos comentado, son muchos los procesador que dan soporte a esta técnica, y disponen en su juego de instrucciones de instrucciones especiales para hacerla posible de una manera sencilla. Algunos de estos procesadores ofrecen instrucciones especiales de precarga de contenidos en la memoria cache, estas instrucciones se ejecutaran durante el arranque del sistema, introduciendo en la cache los contenidos previamente seleccionados y bloqueando la memoria cache una vez acabada dicha precarga. Aunque la mayoría de procesadores optan por sencillas instrucciones *lock/unlock* que permiten bloquear y desbloquear la memoria cache una vez utilizadas, gracias a estas dos instrucciones y al uso de instrucciones *load* para cargar contenidos en la memoria cache podemos utilizar esta técnica en estos procesadores.

Pero aunque bloquear la memoria cache sea sencillo no quiere decir que esta sea una técnica trivial, la verdadera dificultad de esta técnica reside en seleccionar los contenidos que deben permanecer bloqueados en la memoria cache y que hacen que no se vean excesivamente perjudicadas las

prestaciones del sistema. Son muchos los estudios que han abordado este tema en los últimos años, algunos de ellos ya han sido comentados en el apartado de artículos relacionados de esta misma memoria. Algunos de estos estudios utilizan algoritmos genéticos para la selección de contenidos que deben bloquearse en la cache como [23] y [37], otros como [41] utilizan algoritmos de baja complejidad que buscan minimizar el tiempo de CPU necesario para la ejecución de todas las tareas, y otros como [24] se basan en análisis estáticos de todos los accesos a la memoria cache para encontrar las distintas instrucciones que hacen que el funcionamiento de la memoria en el peor de los casos (Worst Case Memory Performance,WCMP) sea igual al que se habría obtenido sin que los contenidos de la cache permanezcan fijos, en [25] aplica esta técnica a un sistema multitarea particionando la cache entre las diferentes tareas, por poner algunos ejemplos.

Como hemos comentado son muchos los procesadores dedicados a sistemas empotrados que dan soporte a esta técnica por ejemplos la familia de procesadores ARM9, ARM11 y Cortex A8s, procesadores empotrados de ARM utilizados para PDAs, teléfonos móviles como el iPhone[26] y algunos sistemas de videojuegos portátiles como la Nintendo DS[27] o Pandora[28], explican en su documentación[31] como dar soporte a dicha técnica. En la documentación también indican que debe evitarse el uso de esta tecnica si el procesador dispone de SPM o como los de ARM prefieren denominar Tightly Coupled Memory (TCM), ya que seria desperdiciar la SPM con la que se obtendría el mismo rendimiento.

La familia de procesadores ARM9 permite el bloqueo de contenidos en cache de primer nivel tanto en la cache de datos como en la instrucciones como en el procesador ARM946E-S [29]. Este procesador tiene una jerarquía de memoria con un primer nivel compuesto por una memoria cache de datos y una cache de instrucciones de tamaño variable entre 0KB y 1MB, estas memorias caches son siempre asociativas de 4 conjuntos y con un tamaño de linea de 8 palabras . El espacio mínimo que se puede bloquear en la memoria cache es de un conjunto, con una granularidad de un conjunto empezando siempre por el conjunto 0 de memoria cache, y pudiendo abarcar hasta los cuatro conjuntos. Ademas antes de bloquear la cache se deben cumplir una serie de condiciones, como por ejemplo que el código a bloquear no este aun dentro de la memoria cache o que el espacio de la cache a bloquear este limpio. Para bloquear código en la cache se debe ejecutar una pequeña rutina software que bloquee la cache programando una serie de registros y después forzando el rellenando de la cache con instrucciones de load LDR.

La familia de procesadores ARM11s y ARM Cortex A8s también proporcionan soporte a esta técnica aunque con ciertas restricciones. Los ARM11s solo pueden bloquear la cache de instrucciones de nivel 1 mientras que en los Cortex A8s solo puede bloquearse la cache unificada de nivel 2. En ambos casos la granularidad mínima que debe bloquearse es un conjunto, generalmente un cuarto de la capacidad total de la memoria. En ambos casos para bloquear código en la memoria cache deben prelanzarse las instrucciones que deben ser bloqueadas en la memoria cache y a continuación bloquear los conjuntos de la memoria que deben ser fijados, liberando al resto tal como indican la ayuda de la documentación proporcionada por ARM [30].

5.1 - SISTEMA SIMULADO

El simulador utilizado en esta memoria, Vatios[14], no incluye por defecto la posibilidad de utilizar memorias cache con bloqueo de contenidos, sin embargo dado que es de código libre y fácilmente modificable es posible añadir los cambios en el código necesarios para poder utilizar esta técnica. En esta memoria se ha modificado el código del simulador y se han añadido las opciones

pertinentes para poder utilizar dicha técnica de una manera sencilla. Las características del sistema de memoria cache con bloqueo implementado en el simulador se relatan a continuación.

En este proyecto se ha optado por implementar esta técnica de una manera sencilla. Se decidirá al principio de la ejecución si la memoria cache estará bloqueada o no y esta permanecerá bloqueada durante toda la ejecución del sistema. Se bloqueara toda la memoria cache y solo se da soporte al uso estático de esta técnica, es decir una vez cargados los contenidos en la memoria cache estos permanecerán invariables durante toda la ejecución. Además se ha decidido que la posibilidad de bloquear las memorias cache sea posible solo en caches de primer nivel, algo lógico teniendo en cuenta que la mayoría de los procesadores destinados a sistemas empotrados carecen de varios niveles de memorias cache y son a este tipo de procesadores a los que va destinado este proyecto.

Para que las prestaciones del sistema no se vean excesivamente lastradas, se utilizará, siempre que la cache este este bloqueada, un pequeño buffer. Este pequeño buffer del tamaño de una sola línea de cache tiene como objetivo evitar la caída de prestaciones debido al principio de localidad espacial. Este buffer funcionará como una memoria cache de una sola línea por lo que todo código que no este dentro del contenido de la cache bloqueada y se ejecute de manera secuencial no supondrá un aumento del tiempo necesario para su ejecución. El sistema obtenido se muestra en la figura.

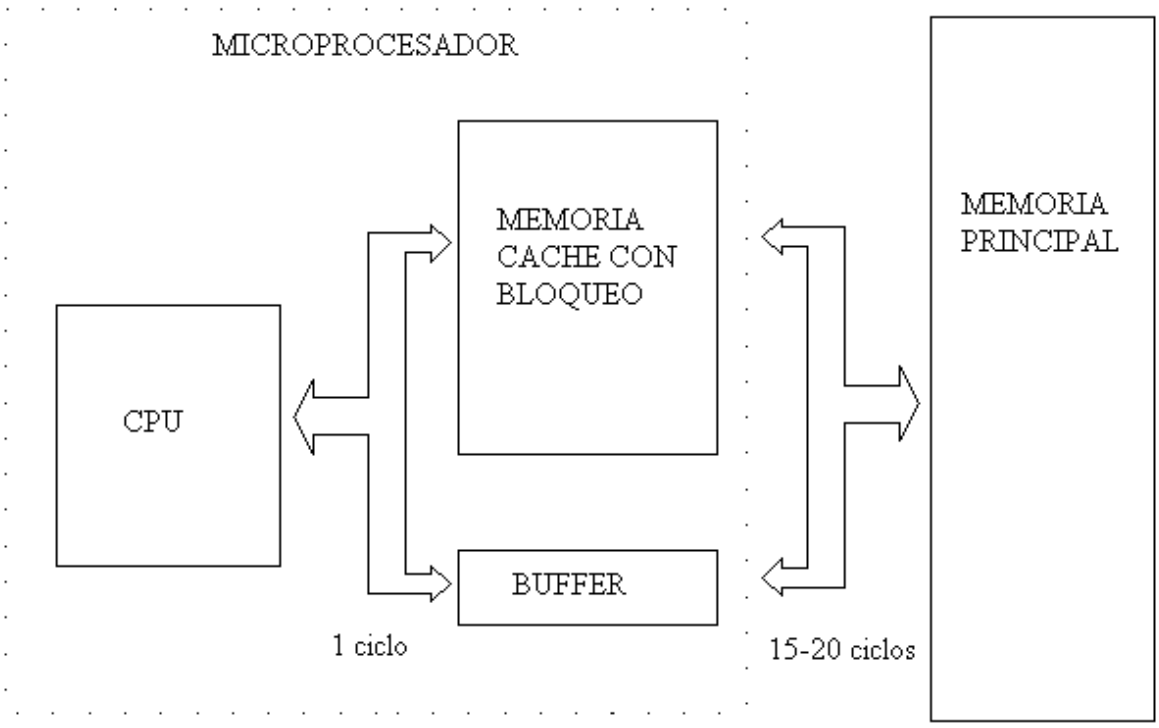

Figura 13 : Sistema memoria cache con bloqueo simulado

El sistema de carga de instrucciones en la cache bloqueada se realiza de la siguiente manera. Al principio de la simulación se pasa como parámetro un fichero que contendrá las direcciones de todas las instrucciones que deben cargarse en la cache, y estas se cargaran en la cache la primera vez que sean referenciadas y permanecerán en la cache el resto del tiempo de ejecución del sistema. Los pormenores relativos a este fichero se explicaran con mas detalle en el siguiente apartado.

El sistema resultante, mostrado en la figura 13 funcionará de la siguiente forma. Cada vez que la CPU referencie una dirección de memoria, la buscará de manera paralela, tanto en la cache como en el buffer. En caso de acierto alguna de estas dos memorias le suministrara la instrucción o dato mientras que en caso de que no se encuentre en ninguna de estas dos memorias, comprobará si debe forzar su carga en la memoria cache o debe cargarla en el buffer mientras busca la linea de cache correspondiente en la memoria principal. Una vez cargada la línea de cache en su estructura correspondiente se le suministrará a la CPU la instrucción o dato y continuará la ejecución.

5.2 - OPCIONES AÑADIDAS AL SIMULADOR

En este apartado explicaremos las distintas opciones añadidas al simulador así como su funcionamiento para poder simular correctamente sistemas que contengan memorias cache con bloqueo de contenidos.

Se ha añadido una ultima opción a la hora de definir una memoria cache. Esta opción definirá si queremos que la cache este o no bloqueada, por defecto no estará bloqueada. Como vemos a continuación la opción <lock> decide si queremos que este bloqueada o no la cache, tiene dos posibles opciones, con una 'l' marcamos el bloqueo de la cache, mientras que con una 'u' marcamos que la cache no este bloqueada. Como podemos observar esto se hace al configurar el sistema que queremos simular por lo que la memoria cache permanecerá bloqueada o no bloqueada durante la totalidad del tiempo que ocupe la simulación.

```
The cache config parameter <config> has the following format:

<name>:<nsets>:<bsize>:<assoc>:<repl>:<lock>

<name>   - name of the cache being defined
<nsets>  - number of sets in the cache
<bsize>  - block size of the cache
<assoc>  - associativity of the cache
<repl>   - block replacement strategy, 'l'-LRU, 'f'-FIFO, 'r'-
random
<lock>   - lock state of the cache, 'l' - locked, 'u' - unlocked

Examples:   -cache:dl1 dl1:4096:32:1:l:u
```

La segunda opción que hemos añadido al simulador sirve para cargar con contenido una memoria cache bloqueada. Esta opción se utiliza para introducirle al simulador un fichero con las direcciones de memoria que queremos bloquear en la memoria cache.

```
-load_cache: <file>   file to load lock cache
```

El formato del fichero debe simplemente incluir las direcciones de memoria que queremos que se se introduzcan en la memoria cache. Las direcciones deberán estar ordenadas de menor a mayor prioridad, estando las de mayor prioridad en las ultimas filas del fichero. No importará que haya varias direcciones de una misma linea de cache, ni que por razones de capacidad o de conflicto en un conjunto, haya mas direcciones de las que pueda contener la memoria cache definida para ser

bloqueada, al final la memoria cache acabara conteniendo las lineas de cache de las instrucciones que han sido marcadas como mas prioritarias, es decir las que estén mas abajo en el fichero y que no tengan conflictos por capacidad o conflictos en un conjunto con otras mas prioritarias.

```
formato del fichero:

direccion1
direccion2
. . . .
```

El fichero se puede hacer de una manera sencilla utilizando el simulador sim-profile. Sim-profile es un simulador que incluye Simplescalar que proporciona gran cantidad de información estadística de los programas que estamos simulando. Utilizando la opción -taddrprof podemos conseguir una estadística de la direcciones de las instrucciones ejecutadas y la cantidad de veces que se han ejecutado en el programa simulado, que de manera sencilla, mediante un pequeño programa en C realizado para tal fin podemos convertir en el fichero que necesitamos. El código del programa para convertir la información que proporciona sim-profile en nuestro fichero de direcciones se incluye en el anexo.

```
./sim-profile -taddrprof <programa a utilizar>
```

5.3 - CAMBIOS INTRODUCIDOS EN EL CÓDIGO DEL SIMULADOR

En este apartado explicamos los cambios y añadidos que hemos realizado en el código del simulador. El diseño de nuestra solución consiste en crear una estructura de cache auxiliar del mismo tamaño y de la misma arquitectura de la cache que queremos bloquear, una vez creada cargamos todas las direcciones del fichero de carga en esta cache por orden de aparición en el fichero. En este momento esta cache auxiliar no sera mas que una imagen de como acabara la cache real que queremos bloquear al final de la ejecución del sistema. Por ultimo en cada lanzamiento de una instrucción, antes de buscar la instrucción en la cache real, la buscamos en la cache auxiliar y en caso de que este en esta cache, marcamos dicha línea de cache para que se fuerce su carga en la cache real y la borramos de la cache auxiliar. Hay que destacar que esta cache auxiliar se utiliza solo a efectos de hacer la carga en la cache real de manera sencilla, no existe en la arquitectura real y ni perjudica a la latencia en la búsqueda de instrucciones ni al consumo energético, simplemente existe a nivel de código. A continuación explicamos los añadidos y cambios realizados en cada uno de los ficheros del simulador.

5.3.1 - FICHERO sim-vatios.c

CREAR CACHE AUXILIAR

En la función *sim_check_options()* creamos la cache auxiliar, esta cache solo la crearemos en el caso de que la cache haya sido definida como una cache bloqueada. Utilizamos los mismos argumentos con los que se han definido la cache bloqueada para crear esta cache auxiliar:

```
/* check simulator-specific option values */
void
sim_check_options(struct opt_odb_t *odb,        /* options
```

```
database */
            int argc, char **argv)        /* command line arguments
*/
{

(...)

//CACHE AUXILIAR
      if(cache_char2state(s)==lock)
      cache_auxiliar = cache_create("auxiliar", nsets, bsize, /*
balloc */FALSE,
                        /* usize */0, assoc, cache_char2policy(c),
                        il1_access_fn, /* hit lat
*/cache_il1_lat,cache_char2state(s));

(...)

}
```

CAMBIOS EN RUU_FETCH

La función *ruu_fetch()* se encarga de simular la primera etapa del procesador segmentado, la etapa de lanzamiento de instrucciones. En esta etapa comprobaremos si la dirección esta dentro de la memoria cache con la función *cache_probe()* y en caso de que así suceda se forzara la carga en la cache de instrucciones, utilizando la variable *meter_direccion* y borramos la línea de cache de esta dirección de la cache auxiliar.

```
/* fetch up as many instruction as one branch prediction and one
cache line acess will support without overflowing the IFETCH ->
DISPATCH QUEUE */
static void
ruu_fetch(void)
{

(...)


          if(cache_il1->state == lock  &&
cache_probe(cache_auxiliar,IACOMPRESS(fetch_regs_PC))!=0){
              //Esta direccion de memoria hay que meterla en cache
              meter_direccion=1;
               //quitamos esta direccion de la cache auxiliar una
vez metida en la cache

cache_load_flush_addr(cache_auxiliar,IACOMPRESS(fetch_regs_PC));
          }

           /* access the I-cache */
```

```
          lat =
        cache_access(cache_il1, Read, IACOMPRESS(fetch_regs_PC),
                  NULL, ISCOMPRESS(sizeof(md_inst_t)), sim_cycle,
                  NULL, NULL,meter_direccion);

 //volvemos a poner la variable a 0 para procesadores
superscalares
          meter_direccion=0;

(...)
}
```

CAMBIOS EN SIM_MAIN

Sim_main() contiene la inicialización del simulador, así como el bucle principal que ejecuta cada una de las etapas del simulador, en este caso simplemente añadimos código para que se cargue el fichero de direcciones en la cache auxiliar, con la ejecución de la función *cache_load()* que explicaremos mas tarde, antes de que se inicie el bucle principal del simulador.

```
/* start simulation, program loaded, processor precise state
initialized */
void
sim_main(void)
{
(...)


//CARGAMOS FICHERO EN CACHE AUXILIAR
  if(cache_il1 != NULL && cache_il1->state==lock &&
strcmp(file_load_cache,"no file")!=0){
      cache_load(cache_auxiliar,file_load_cache);
  }

(...)
}
```

5.3.2 - FICHERO cache.h

AÑADIMOS NUEVAS CARACTERISTICAS A LA ESTRUCTURA CACHE_T

La estructura cache_t define la estructura de cada una de las memorias cache que utiliza el simulador. Para hacer posible el bloqueo de cache hemos tenido que añadir una serie de variables. La enumeración *cache_state* marca si la cache esta o no bloqueada, *last_tagset_buffer* guarda la etiqueta de la línea de cache que esta alojada en el pequeño buffer, explicado con anterioridad en esta la memoria, utilizado para mejorar las prestaciones del sistema en caso de utilizar una cache bloqueada. Por ultimo la variable *accesos_buffer* simplemente guarda las estadísticas de los cambios de líneas de cache que se producen en el buffer.

```
/* cache definition */
```

```
struct cache_t
{
  /* parameters */
(...)
  enum cache_state state;     /* cache state locked or unlocked*/
(...)
  md_addr_t last_tagset_buffer; /* last line tag in the temporal*/
(...)
  unsigned int accesos_buffer; /*cambios en el buffer temporal */
(...)
};
```

5.3.3 - FICHERO cache.c

CAMBIOS EN CACHE_CREATE

La función *cache_create()* se encarga de crear e inicializar cada estructura *cache_t*. En esta función simplemente añadimos unas lineas de código para que inicialice el estado de la cache, el buffer y las estadísticas de los accesos a este.

```
/* create and initialize a general cache structure */
struct cache_t *              /* pointer to cache created */
cache_create(char *name,      /* name of the cache */
          int nsets,          /* total number of sets in cache */
          int bsize,               /* block (line) size of cache */
          int balloc,         /* allocate data space for blocks? */
          int usize,     /* size of user data to alloc w/blks */
          int assoc,               /* associativity of cache */
          enum cache_policy policy,     /* replacement policy w/in
sets */
          /* block access function, see description w/in struct
cache def */
          unsigned int (*blk_access_fn)(enum mem_cmd cmd,
                        md_addr_t baddr, int bsize,
                        struct cache_blk_t *blk,
                        tick_t now),
          unsigned int hit_latency,     /* latency in cycles for a
hit */
          enum cache_state state) /* state of the cache */
(...)

 cp->state = state;

(...)

 cp->last_tagset_buffer = 0;
 cp->accesos_buffer = 0;
```

```
(...)
}
```

CAMBIOS EN CACHE_ACCESS

La función *cache_access()* es la encargada de controlar los accesos a memoria cache, esta función devuelve la latencia que tarda un acceso a memoria cache y actualiza las etiquetas de las líneas que están en la memoria cache en caso de fallo. En esta función debemos añadir el código necesario para controlar los accesos en el caso de que la memoria cache este bloqueada, para que busque los datos en el buffer y en la cache y actualice el buffer o fuerce su entrada en la cache.

```
/* access a cache, perform a CMD operation on cache CP at address
ADDR,
   places NBYTES of data at *P, returns latency of operation if
initiated
   at NOW, places pointer to block user data in *UDATA, *P is
untouched if
   cache blocks are not allocated (!CP->BALLOC), UDATA should be
NULL if no
   user data is attached to blocks */
unsigned int                  /* latency of access in cycles */
cache_access(struct cache_t *cp,   /* cache to access */
          enum mem_cmd cmd,        /* access type, Read or Write */
          md_addr_t addr,          /* address of access */
          void *vp,           /* ptr to buffer for input/output */
          int nbytes,         /* number of bytes to access */
          tick_t now,         /* time of access */
          byte_t **udata,          /* for return of user data ptr
*/
          md_addr_t *repl_addr,    /* for address of replaced block
*/
          int forzar_insertado)     /* fuerza a que esta direccion
se meta en la cache- solo para caches bloqueadas*/

{
(...)

//VEMOS SI LO ENCUENTRA EN EL BUFFER TEMPORAL
  if(CACHE_TAGSET(cp, addr) == cp->last_tagset_buffer){

      blk = NULL;
      goto cache_hit;
  }

(...)

 /* update block tags */
  if(cp->state==unlock || forzar_insertado ==1){
```

```
//(CODIGO PARA ACTUALIZAR LAS ETIQUETAS DE LA MEMORIA CACHE)

  }
  else{
      //Cambio de linea de cache en el buffer
      //Actualizar etiqueta
      cp->last_tagset_buffer = CACHE_TAGSET(cp, addr);
      //Contabilizar cambio en el buffer
      cp->accesos_buffer++;

(...)

  }


(...)
}
```

NUEVAS FUNCIONES

Hemos añadido una serie de nuevas funciones para hacer posible de manera sencilla el bloqueo de contenidos en una memoria cache.

CACHE_STATE

Esta nueva función se encarga simplemente de convertir un carácter a una enumeración sobre el estado de la memoria cache, para poder operar de manera sencilla con el estado de la cache.

```
/*
Pasar de char a estado de la cache
*/
enum cache_state              /* replacement state enum */
cache_char2state(char c)      /* replacement state as a char */
{
 switch (c) {
  case 'l': return lock;
  case 'u': return unlock;
  default: fatal("bogus cache state, `%c'", c);
  }

}
```

CACHE_LOAD_ADRR

Esta función simplemente carga una linea en la memoria cache. Se utilizara cuando vayamos cargado cada una de las lineas del fichero de direcciones.

```
void
```

```
cache_load_addr(struct cache_t *cp,     /* cache to access */
          md_addr_t addr)          /* address to load */
{

    md_addr_t tag = CACHE_TAG(cp, addr);
    md_addr_t set = CACHE_SET(cp, addr);
    md_addr_t bofs = CACHE_BLK(cp, addr);
    struct cache_blk_t *repl;

    /* select the appropriate block to replace, and re-link this
entry to the appropriate place in the way list */

    // case LRU: y case FIFO:
    repl = cp->sets[set].way_tail;
    update_way_list(&cp->sets[set], repl, Head);

    /* remove this block from the hash bucket chain, if hash
exists */
    if (cp->hsize)
     unlink_htab_ent(cp, &cp->sets[set], repl);

    /* update block tags */
    repl->tag = tag;
    repl->status = CACHE_BLK_VALID;     /* dirty bit set on update
*/

  /* link this entry back into the hash table */
    if (cp->hsize)
     link_htab_ent(cp, &cp->sets[set], repl);

}
```

CACHE_LOAD

La función *cache_load()* carga en una memoria cache las direcciones que contiene el fichero que le introducimos como parámetro ayudándonos de la función antes descrita *cache_load_addr()*.

```
void
cache_load(struct cache_t *cp,     /* cache to access */
        char *fichero)  /*fichero a cargar*/
{
    FILE *fich;
    md_addr_t addr=0;

    fich=fopen(fichero,"r");
    if(fich==NULL){
     fatal("I can't open the file, `%s'", fichero);
    }
```

```
    while(fscanf(fich,"%x",&addr)!=EOF){
     if(cache_probe(cp,addr)==0){
         cache_load_addr(cp,addr);

     }
    }

    fclose(fich);

}
```

CACHE_LOAD_FLUSH_ADDR

Por ultimo la función *cache_load_flush_addr()* borra de la memoria cache la línea de cache de la dirección que le pasamos como parámetro.

```
void
cache_load_flush_addr(struct cache_t *cp,    /* cache to access */
                md_addr_t addr)                 /*addr to flush*/
{
    md_addr_t tag = CACHE_TAG(cp, addr);
    md_addr_t set = CACHE_SET(cp, addr);
    struct cache_blk_t *blk;

    if (cp->hsize)
    {
     /* higly-associativity cache, access through the per-set hash
tables */
     int hindex = CACHE_HASH(cp, tag);

     for (blk=cp->sets[set].hash[hindex];
          blk;
          blk=blk->hash_next)
     {
         if (blk->tag == tag && (blk->status & CACHE_BLK_VALID))
          break;
     }
    }
  else
  {
      /* low-associativity cache, linear search the way list */
      for (blk=cp->sets[set].way_head;
        blk;
        blk=blk->way_next)
      {
       if (blk->tag == tag && (blk->status & CACHE_BLK_VALID))
           break;
      }
```

```
  }
    if (blk)
    {
     blk->tag=0;


      /* move this block to tail of the way (LRU) list */
      update_way_list(&cp->sets[set], blk, Tail);
    }
    else
     fatal("I can't find in the cache auxiliar the tag 0x, `
%08x'",addr);

}
```

CAPÍTULO 6 : SCRATCH PAD MEMORY

Una Scratch pad memory (SPM) es un tipo de memoria on-chip SRAM, de semejante utilidad que una memoria cache. Esta memoria posee una serie de características que la hacen especialmente útil en el campo de los sistemas empotrados y los sistemas de tiempo real. En comparación a las memorias cache, las SPM son memorias mas sencillas, ya que requieren de menos lógica de control , mas rápidas, con un menor consumo energético, además de ser totalmente deterministas.

Las SPM son memorias on-chip mapeadas dentro del rango de direcciones accesibles por la CPU de la misma manera que la memoria principal. Al estar mapeadas dentro del rango de direcciones accesibles por el procesador no requieren de etiquetas ni de toda la lógica de control encargada de compararlas, lo que reducirá el consumo energético y el coste de estas memorias frente las memorias cache y aumentara la rapidez con la que se suministra el contenido a la CPU al no tener que hacer comparaciones. Sin embargo, esto también conlleva una desventaja, ya que la sencillez de estas memorias hace que pierda el comportamiento automático del que disponen las memorias cache, siendo necesario introducir instrucciones para copiar datos o instrucciones a las direcciones de memoria mapeadas en la SPM. Esto puede llevarse a cabo de distintas maneras, como veremos mas tarde en los distintos estudios realizados sobre estas memorias, siendo el programador el encargado de analizar el sistema para introducir, añadiendo ciertas instrucciones al programa, las instrucciones o datos en la SPM que optimicen el rendimiento del sistema.

En la figura 14 podemos ver una comparación de la arquitectura de las dos memorias. Como podemos observar la SPM prescinde de buena parte de la lógica utilizada por la cache, como el array de etiquetas o toda la lógica destinada a la comparación de estas, siendo una memoria mucha mas sencilla lo que repercute en un menor tamaño en el silicio del procesador, menos consumo energético y una mayor rapidez.

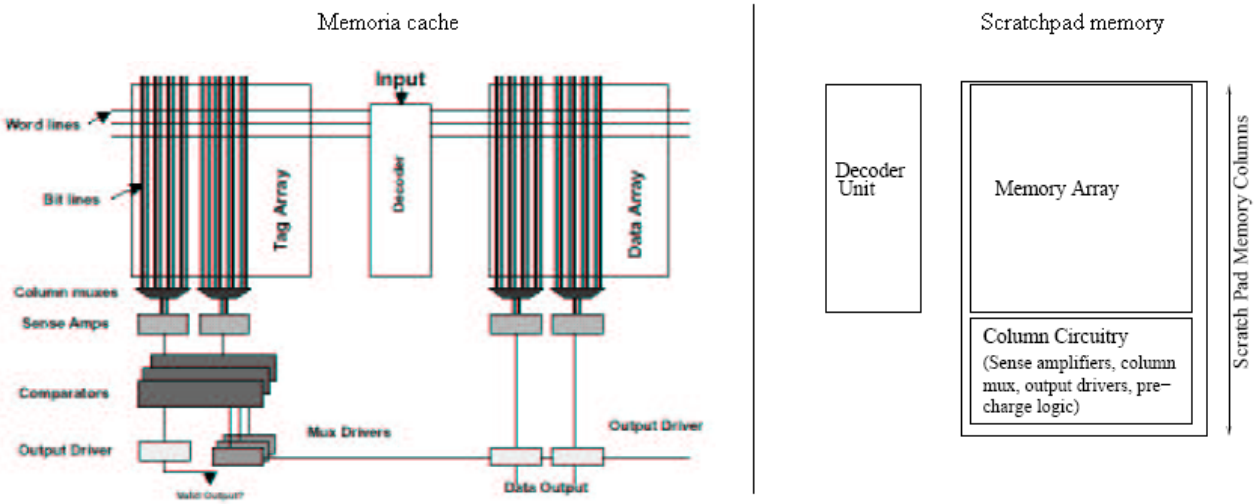

Figura 14 : Comparación organización memoria cache vs organización SPM.

Como hemos comentado mientras que las transferencias de datos a la memoria cache son gestionadas por su controlador y transparentes a la CPU, en la SPM necesitamos hacerlas manualmente mediante instrucciones. Existen dos alternativas, que la CPU se encargue de todo el proceso o utilizar una DMA que ayude a liberar a la CPU de esa tarea. La forma mas sencilla de realizar las transferencias es incluyendo en el código instrucciones load/store, de esta forma la CPU

se encarga de realizar la transferencia de datos a la SPM ejecutando dichas instrucciones. Este método no requiere de hardware adicional siendo el método mas económico y que a priori menor consumo energético presenta, aunque el rendimiento de la CPU puede resentirse debido a el aumento de instrucciones ejecutadas por esta. La otra opción consiste en añadir un DMA al hardware del sistema. Un DMA (Direct Memory Access, acceso directo a memoria) permite escribir o leer de memoria principal de manera independiente de la CPU, de esta manera podemos escribir datos e instrucciones a la SPM desde memoria principal sin que la CPU se quede bloqueada al hacerlo, pudiendo, mientras se realiza dicha transferencia, seguir ejecutando otras instrucciones. Todo DMA consta principalmente de un controlador y de una unidad de transferencias, el controlador actúa de interfaz entre la CPU y la DMA y cuando recibe una petición de transferencia por parte de la CPU consulta el estado de la unidad de transferencias y inicia la transferencia en caso de que esta este desocupada. Mientras que la unidad de transferencias es la encargada de realizar las transferencias entre la memoria principal y la SPM. Añadir una DMA es una opción utiliza en muchos sistemas con SPM que permite mejorar el rendimiento de un sistema con dicha memoria y es altamente recomendable en sistemas empotrados de alto rendimiento. Sin embargo complica en cierta medida el hardware del sistema desaprovechando en parte el menor consumo energético y tamaño en silicio que presenta una SPM frente a una memoria cache.

Son muchos los sistemas empotrados que optan por incluir una SPM en sus arquitecturas, debido a las ventajas que ofrece en forma de determinismo en el código ejecutado, menor consumo energético y menor coste económico, cualidades muy a tener en cuenta en estos sistemas. Sin embargo una mala elección en el código introducido en la SPM puede dar al traste con todas estas ventajas, obteniendo una sensible degradación del rendimiento del sistema. Es por ello que en los últimos años son muchos los estudios que tratan de encontrar la mejor solución a este problema.

De la misma manera que en la asignación de código en una memoria cache con bloqueo, los distintos estudios sobre asignación de código en una SPM pueden dividirse en dos clases, la asignación de código de manera estática donde los contenidos son introducidos al principio de la ejecución y permanecen invariables durante toda la ejecución del sistema, o de manera dinámica donde los contenidos de la SPM varían durante la ejecución del sistema. Algunos de los estudios que utilizan la SPM de manera estática son [1], [2] [3] y [4]. En [1] Banakar nos presenta una comparación entre una SPM y una memoria cache en términos de área, eficiencia energética, además de utilizar un sencillo algoritmo basado en el problema de la mochila para elegir de manera estática en la fase de diseño, el código a introducir en la SPM para realizar las simulaciones que muestra en los resultados. En [2] Verma se basa en la creación de un grafo de conflictos, donde cada vértice contiene un bloque básico de memoria conectados por los bloques básicos por los que alguna de sus lineas de cache son substituidas según la política de remplazamiento en caso de fallo de cache. Con ayuda de este grafo de conflictos crea un problema de programación entera con el cual seleccionar el subconjunto de bloques básicos que minimizan el numero de aristas de conflicto y el consumo energético de sistema, que son los bloques que introducirá de manera estática en la SPM. En [3] Verma simplemente mejora esta técnica introduciendo un entorno multitarea y tres posibles estrategias de partición del espacio de la SPM entre los procesos implicados, una en la que el proceso activo se hace con todo el espacio de la memoria de la SPM, una segunda donde el espacio es repartido uniformemente entre todas las tareas, y una mixta que junta ambas estrategias. En todas estos estudios es necesario conocer el tamaño de la SPM en tiempo de compilación. En [4] sin embargo Nguyen et al. Nos presenta una técnica de asignación estática donde no es necesario conocer en tiempo de compilación el tamaño de la SPM. Esto es posible a que guarda en el binario cierta información de profiling que utiliza para retrasar la decisión de los objetos que

deben incluirse en la SPM hasta la carga de la aplicación.

Algunos ejemplos de estudios sobre la asignación de código de manera dinámica son [5], [10], [12] y [21]. En [21] Verma et al. utiliza un grafo de control de flujo (CFG) obtenido a partir de las estadísticas de los fallos/aciertos de la cache para mediante la formulación de un problema de programación lineal entera obtener el conjunto de objetos de memoria que debe contener en cada momento la scratchpad. En [5] Egger et al. nos presenta una técnica donde introduce trozos de código en la SPM en tiempo de ejecución bajo demanda, tanto del código programa ejecutado como de las librerías que utiliza, que son elegidos mediante la realización y resolución de un problema de programación lineal entera. En [10] Steinke et al. utiliza un algoritmo que analiza el código de la aplicación a ejecutar identificando sus partes o bloques básicos de código, eligiendo los mejores bloques básicos que deben ser copiados en la SPM mediante la resolución de un problema de programación lineal entera e introduciendo de la mejor manera posible las instrucciones para copiar estos trozos de código en la SPM antes de su ejecución. Por ultimo en [12] L.Li et al. con ayuda de un grafo de control de flujo, un análisis de los rangos de vida de los distintos objetos de memoria y un algoritmo adaptado del generalizado algoritmo de coloración de grafos considera el mejor conjunto de arrays de datos y el mejor instante en que deben copiarse en la SPM.

Son muchos los procesadores diseñados para sistemas empotrados que optan por incluir una SPM en sus diseños, tanto sustituyendo a las memorias cache como complementándolas. Los microprocesador de Freescale [32] , anteriormente el sector dedicado a semiconductores de Motorola, utilizan en sus microprocesadores/microcontroladores dedicados a sistemas empotrados Coldfire [33], basados en la arquitectura 68K, memorias on-chip SRAM (SPMs) complementado, en toda la gama de microprocesadores, o sustituyendo, en toda la gama de microcontroladores, a memorias cache. Los tamaños de estas scratchpad memories varían entre 512B y 32KB según las distintas gamas de microprocesadores disponibles. Estas memorias se encuentran dentro del tango de direcciones no cacheables y la carga de instrucciones o datos dentro de estas memorias se realiza de forma sencilla mediante instrucciones especialmente diseñadas para mover datos o instrucciones de memoria a memoria. Estas instrucciones están diseñadas para proporcionar el máximo rendimiento y están optimizadas para mover varias lineas de memoria mediante ráfagas entre la memoria principal y la SPM con la ejecución de una sola instrucción.

La familia de procesadores de ARM, ARM9, ARM11 y Cortex [34] ofrecen en sus diseños memorias configurables on-chip denominadas Tightly-coupled memory (TCM) que pueden actuar como una SPM, en el primer nivel de la jerarquía de memoria junto a memorias cache de instrucciones y de datos. Muchos de estos procesadores las separan en TCM de datos y TCM de instrucciones y en algunos casos pueden ser configuradas tanto para actuar como una SPM como para actuar como una Smartcache. Los procesadores de gama mas alta utilizan una DMA, que actúa en paralelo al procesador, para cargar datos e instrucciones de la memoria principal en las TCM, mientras que los de gama mas baja utilizan instrucciones especiales parecidas a las que utiliza los microprocesadores de Freescale [32].

6.1 - SISTEMA SIMULADO

En este proyecto se ha optado por implementar en el simulador un modelo de SPM simple y sencillo, acorde con los sistemas empotrados a los que va destino dicho proyecto. Para ello se ha decidido prescindir de utilizar una DMA, que supondría un mayor coste energético y monetario, aun a costa de reducir en cierta medida las prestaciones del sistema al ser la CPU directamente la

encargada de gestionar la carga de instrucciones en la SPM. En este proyecto se ha puesto especial énfasis en la búsqueda rápida de instrucciones en memoria, debido en buena parte a que los accesos a memoria para la búsqueda de instrucciones representa aproximadamente el 75% de las referencias a memoria generadas por la CPU. Es por ello que nuestra SPM solo se ha implementado contener instrucciones[35].

La carga de instrucciones en la SPM se realiza mediante la inclusión de instrucciones de load y store en el código del programa siendo responsabilidad del programador el cerciorarse si la cantidad de instrucciones que van a introducirse en la SPM no rebasan la capacidad total de la SPM simulada. Para facilitar el uso de la SPM se ha creado una función en C que introduce las instrucciones en ensamblador para cargar fácilmente instrucciones en la SPM, el uso de esta función y sus pormenores se explicara en los siguientes apartados. De esta manera añadiendo las instrucciones de carga en los momentos oportunos se puede utilizar de manera sencilla la SPM para asignar código en la SPM tanto de manera estática como de manera dinámica.

El simulador esta preparado para aceptar cualquier configuración que incluya cualquier tipo de memoria cache de primer nivel con una SPM, y también para funcionar con solo una SPM. Por defecto se ha configurado para que si se configura un sistema con una SPM y sin memoria cache de instrucciones de primer nivel, se habilite, de la misma manera que con la memoria cache con bloqueo, un pequeño buffer, de una sola línea de cache. Este buffer funciona como si de una pequeña memoria cache de una sola línea se tratara, evitando la perdida de prestaciones cuando ejecutamos código secuencial. El sistema obtenido se muestra en la figura .

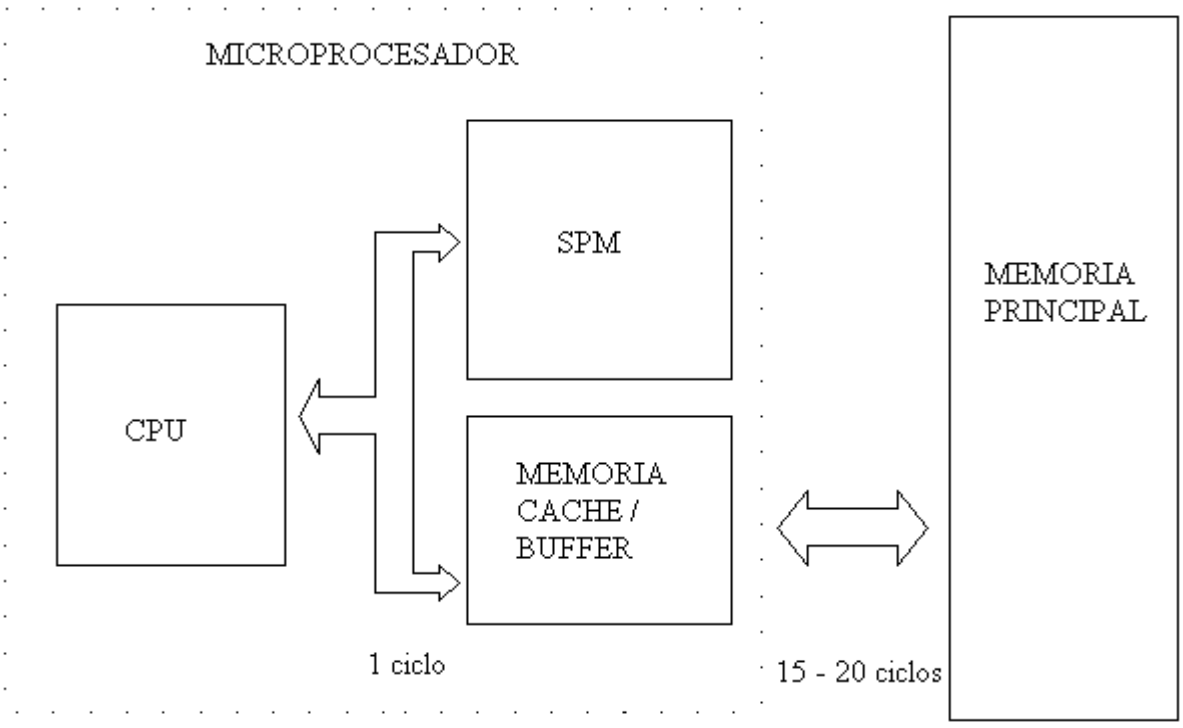

Figura 15: Sistema con SPM implementado.

Como se puede apreciar en la figura 15 la interacción entre la SPM y la memoria principal se realiza siempre a través de la CPU. Es decir mediante instrucciones de load/store tal como hemos explicado

anteriormente.

6.2 - OPCIONES AÑADIDAS AL SIMULADOR

En este apartado explicaremos las opciones que se han añadido al simulador para poder configurar de forma sencilla un sistema con una SPM, así como la forma de introducir de manera mas o menos sencilla el código en un programa en C para poder copiar instrucciones en la SPM. En primer lugar explicaremos las distintas opciones de configuración que se han añadido al simulador para poder simular la SPM deseada.

Se ha añadido una opción para configurar el tamaño del buffer de instrucciones. Por defecto si no se configura ninguna memoria cache de instrucciones de primer nivel y se incluye una memoria SPM, se crea una configuración con un buffer de 32 bytes. Este buffer sera siempre del tamaño de una linea de cache, por lo que verdaderamente nos estamos refiriendo al tamaño de esa linea de cache. Es posible también indicar que queremos que nuestro sistema funcione únicamente con una SPM indicando que queremos un tamaño de buffer de 0 bytes. En caso de configurar un sistema con una memoria cache de instrucciones de primer nivel esta opción no sera tenida en cuenta.

```
-buffer:size   <size of temporal buffer (in bytes)>
```

También se ha añadido una opción para poder configurar el tamaño de la SPM. Un tamaño mayor que 0 indica que el sistema cuenta con una SPM, por defecto el sistema se configura para no contar con una SPM. Para poder trabajar de manera mas sencilla, esta opción esta configurada para recibir el numero de instrucciones que puede contener la SPM, en lugar de su tamaño en bytes. Se ha realizado así para que el programador tenga totalmente claro el numero de instrucciones que caben en la SPM simulada ya que es responsabilidad suya introducir las instrucciones de manera correcta, sin que sobrepasen el tamaño de la SPM. Esta opción en este caso solo funciona a efectos de calcular el consumo energético de la SPM y definir que el sistema dispone de una SPM, nunca a fin de comprobar si se han incluido mas instrucciones en la SPM que el tamaño permitido por esta, siendo esto último responsabilidad del programador.

```
-spm:size      <size of scratchpad memory (in instructions)>
```

Por ultimo se incluye una opción para definir la latencia de la memoria SPM. Se entiende por este valor como el numero de ciclos que tarda esta memoria en devolver la instrucción pedida a la CPU, siendo por defecto uno.

```
-spm:lat       <scratchpad memory latency (in cycles)>
```

Por ultimo explicamos como introducir las instrucciones necesarias para introducir código en la SPM de manera correcta y mas o menos sencilla en un programa escrito en un lenguaje de alto nivel como C. Para ello hemos creado una función para introducir código en la SPM mediante instrucciones de load/store y una serie de instrucciones en ensamblador que deben ejecutarse antes y después del código introducido para que este se ejecute desde la SPM. A continuación lo explicamos con mas detalle.

Para este ejemplo utilizaremos el programa binary search, bs, incluido en los benchmarks The Mälardalen WCET research group [18]. Este programa realiza una búsqueda binaria en un vector de

15 enteros, en la SPM introduciremos el bucle principal del programa, que realiza la búsqueda en el vector.

Para empezar introduciremos en el programa la función spmfill. Esta función esta creada especialmente, mediante instrucciones de load/store en ensamblador, obteniendo como parámetros la dirección inicial y el numero de instrucciones a introducir en la SPM. Ejecutando entonces las instrucciones precisas para introducir el trozo de código deseado en la SPM. Se explicará con mas detalle en el siguiente apartado.

```
void spmfill(int addr, int instr){
    int i=0;

    __asm__ __volatile__("addu $8 ,$0 ,$0");
    __asm__ __volatile__("lui $8 ,0x8000");
    __asm__ __volatile__("or $8 ,$8 ,$4");


    for(i=0;i<(instr+4);i++){
       __asm__ __volatile__("dlw $10 ,0($4)");
       __asm__ __volatile__("dsw $10 ,0($8)");
       __asm__ __volatile__("addiu $4 ,$4 ,8");
       __asm__ __volatile__("addiu $8 ,$8 ,8");
    }

}
```

A continuación introducimos dos instrucciones nop al principio y al final del trozo de código que queremos insertar en la SPM, como podemos ver en la figura 16.

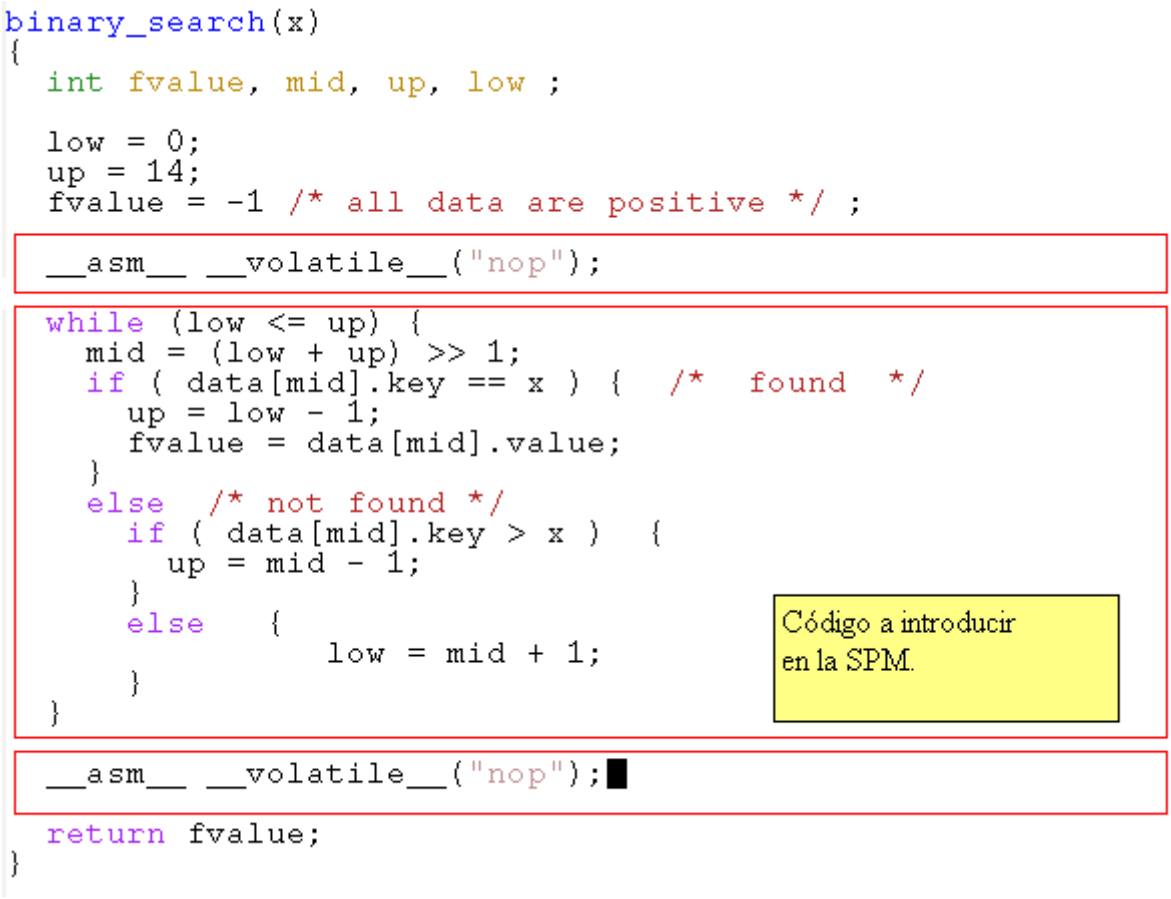

Figura 16 : Paso 1 para la introducción de código en la SPM.

Seguidamente compilamos el programa, mediante el compilar sslittle proporcionando con las herramientas de Simplescalar:

bin/sslittle-na-sstrix-gcc -o bs bs.c

Para a continuación desemsamblar el código resultante con el programa objdump:

bin/sslittle-na-sstrix-objdump -d bs | less

Obtendremos algo semejante a la semejante a la figura 17. Buscando en el código ensamblador obtenido podemos observar las dos instrucciones nop, representadas por el desemsamblador como puntos suspensivos, envolviendo el código a introducir en la SPM. Con las direcciones de la primera y ultima instrucción del segmento de código a introducir podremos calcular el numero de instrucciones que debemos introducir en la SPM.

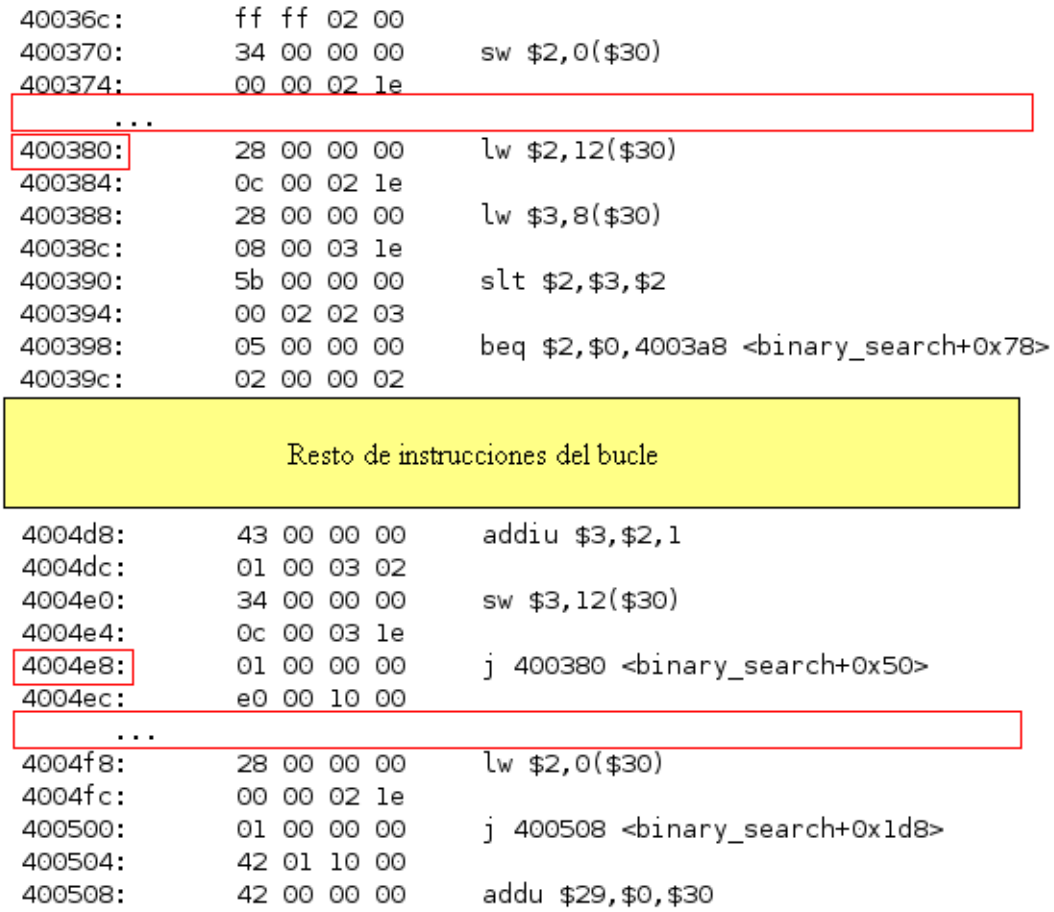

Figura 17 : Paso 2 para la introducción de código en la SPM.

Para calcular el numero de instrucciones a introducir en la SPM, simplemente debemos coger la dirección de la ultima instrucción nop, restarle la dirección de la primera instrucción y dividir el resultado entre 8, que es el numero de bytes que ocupa una instrucción y convertirlo de hexadecimal a decimal. En nuestro caso:

(0x4004e8 + 0x08) – 0x400380 = 0x170

0x170 / 0x08 = 0x2E

Lo pasamos a decimal:

0x2E → 46 instrucciones.

Una vez sabemos el numero de instrucciones que tiene el bucle, podemos introducir la función spmfill en el código en C. Ademas añadimos cuatro instrucciones nop, que posteriormente cuando sepamos con seguridad la dirección inicial del código a introducir en la SPM transformaremos en una instrucción de salto a las direcciones de la SPM. Aprovechamos también para introducir al final del código, cuatro instrucciones nop que después convertiremos en las direcciones de salto de vuelta a direcciones ejecutables en memoria principal o cacheables. Como parámetros de la función spmfill utilizaremos la dirección actual de inicio del código a introducir en la SPM, que sera una

aproximación de la dirección inicial, ya que cambiará después de introducir todas estas nuevas instrucciones, y el numero de instrucciones calculadas en el anterior apartado. Podemos ver el resultado en la figura 18.

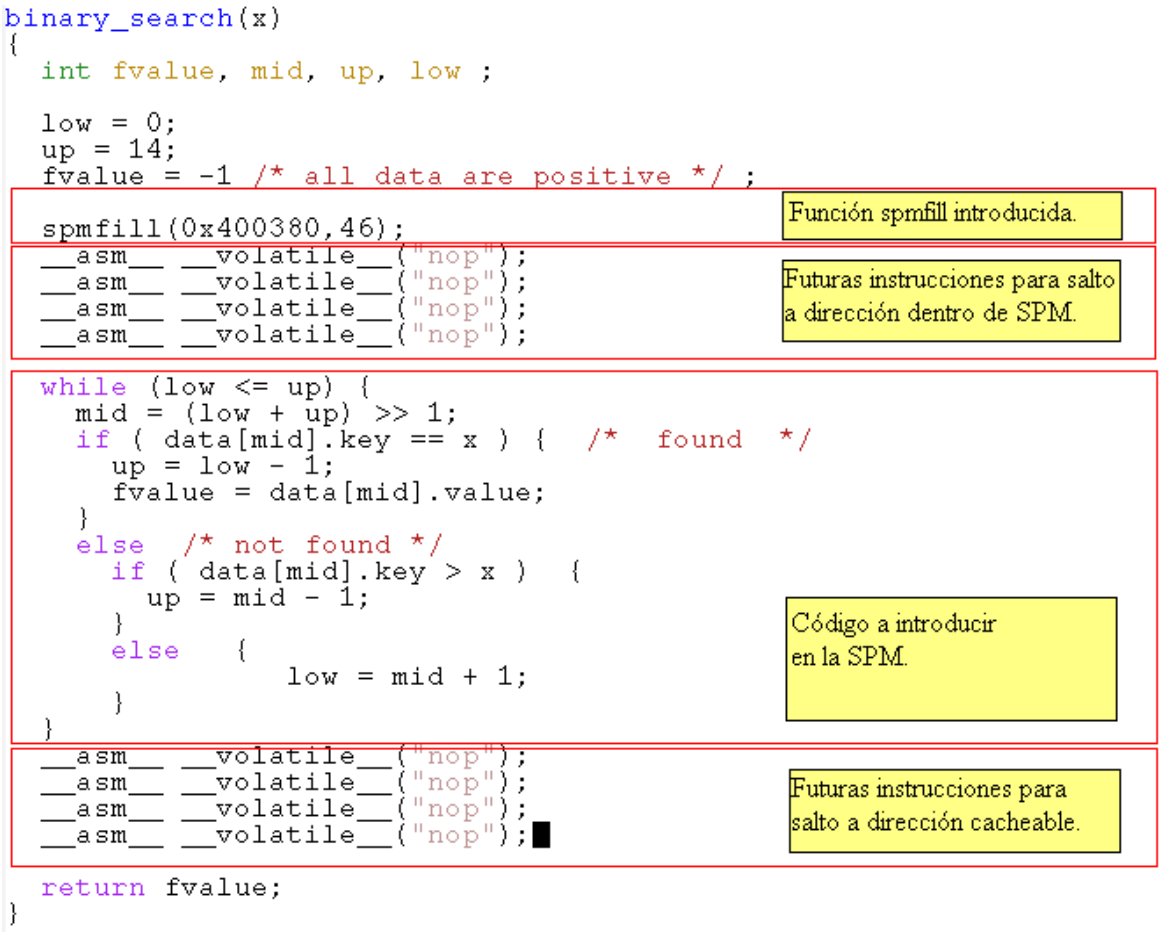

Figura 18 : Paso 3 para la introducción de código en la SPM.

A continuación volvemos a compilar el programa y desensamblamos el programa resultante. Ahora podemos deducir ya exactamente cual sera las dirección de inicio a la memoria SPM y la dirección de vuelta o dirección destino a las direcciones cacheables del programa. En la figura 19 se puede observar el resultado después de compilar y desensamblar el código resultante.

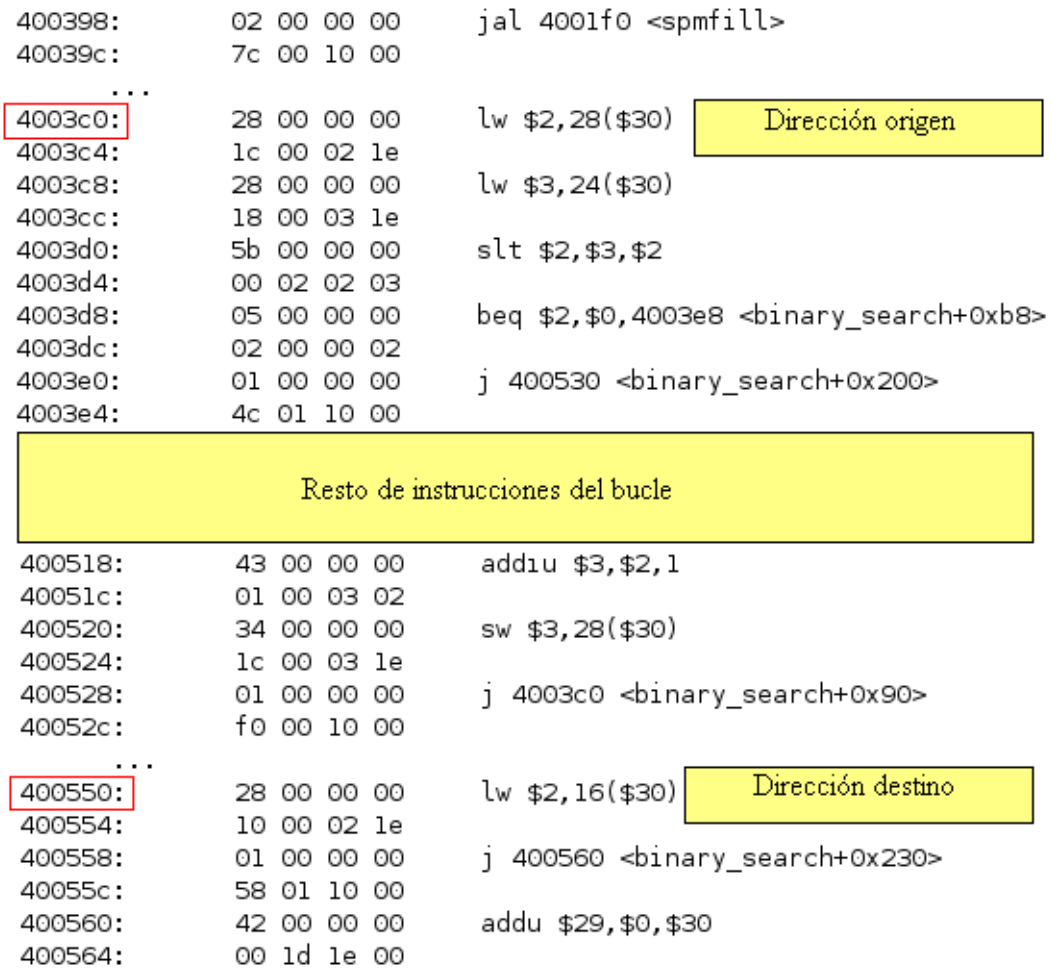

Figura 19 : Paso 4 para la introducción de código en la SPM.

Ahora simplemente tenemos que modificar el programa cambiando las instrucciones nop introducidas por las instrucciones correctas para realizar los saltos. Lo primero que debemos cambiar es la dirección que le pasamos como argumento a la función spmfill, que debido a las nuevas instrucciones nop introducidas habrá cambiado, esta dirección indica el inicio del código a introducir en la SPM. A continuación cambiamos las cuatro instrucciones nop siguientes por cuatro instrucciones para saltar a una dirección entre el rango de direcciones de 0x8000000 y 0x8fffffff, para simplificar y evitar errores en saltos realizados mediante direccionamiento por dirección como podría ser una llamada a función se utiliza la dirección 0x80000000 + dirección de inicio del código a introducir en la SPM. Las direcciones después de la 0x80000000 están reservadas en el simulador para futuros desarrollos y es donde nosotros situaremos la SPM, esto se explicara con mas detalle en el siguiente apartado. Por ultimo cambiaremos las ultimas cuatro instrucciones nop por las instrucciones necesarias para saltar al código que se ejecutara después del código introducido en la SPM, que es la dirección que en la figura . se indica como dirección destino. En la figura 20 se puede ver el código resultante de esta operación.

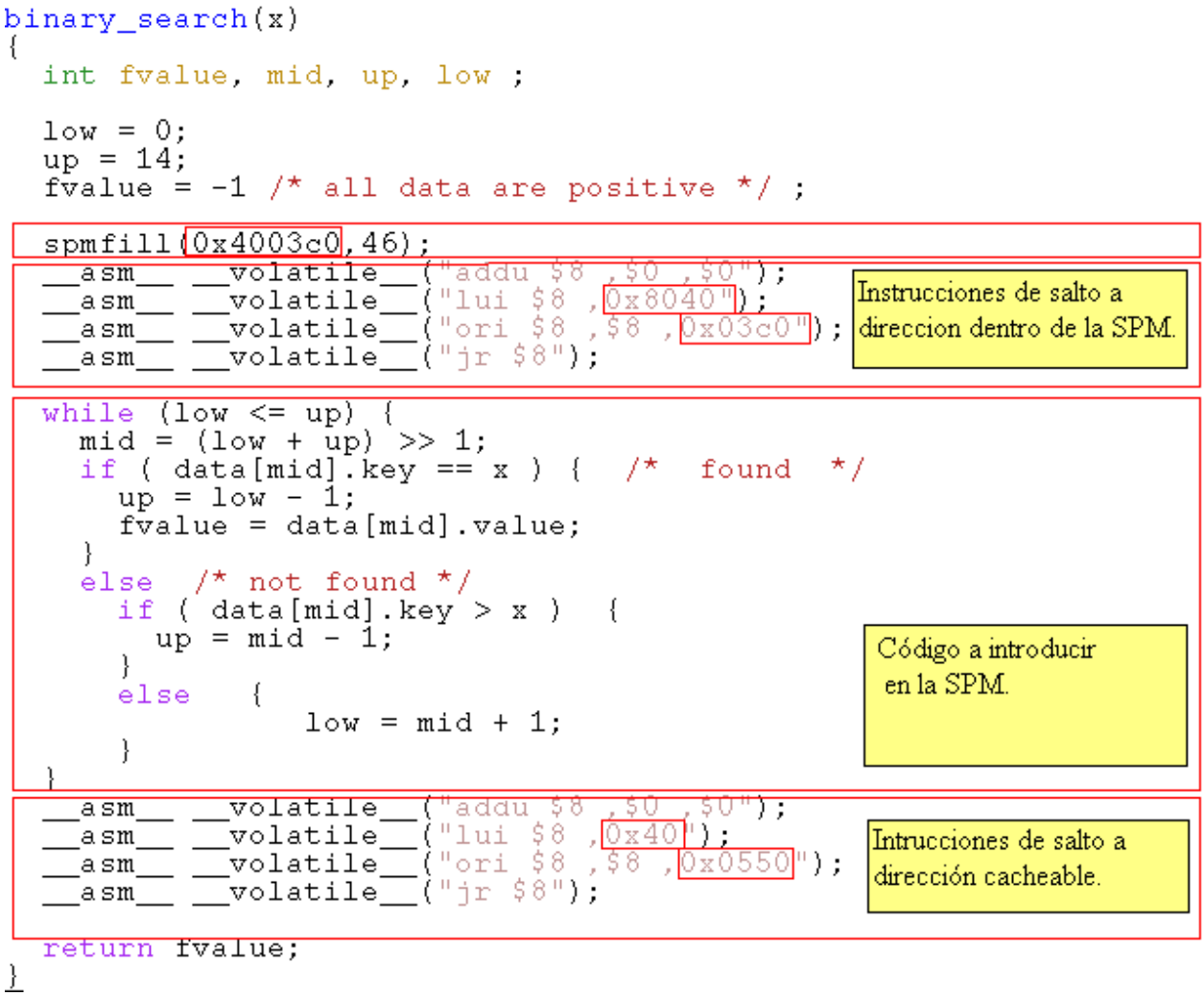

Figura 20 : Paso 5 para la introducción de código en la SPM.

Ya por ultimo simplemente nos queda compilar y ejecutar el programa. Podremos comprobar como el código introducido se ejecuta efectivamente en la SPM.

6.3 - CAMBIOS INTRODUCIDOS EN EL CÓDIGO DEL SIMULADOR

En este apartado explicaremos los cambios y añadidos que se han realizado en el código del simulador para implementar la SPM. También explicaremos el código que permite a los programas de benchmark utilizar la SPM implementada en el simulador. En primer lugar explicaremos el subsistema de memoria del simulador Simplescalar.

6.3.1 - SUBSISTEMA DE MEMORIA DEL SIMULADOR

El espacio de memoria virtual del simulador Simplescalar se muestra en la figura 21 es de 2^{32} bytes, mapeadas en las posiciones del 0x00000000 al 0x7fffffff. El espacio de memoria de 0x00000000 al 0x00400000 esta actualmente sin uso, el espacio de memoria de 0x00400000 a 0x10000000 es usado para alojar el código de los programas ejecutados. Los datos se alojan en a partir de la posición 0x10000000 hasta mem_brk_point. Primero se guardan en el segmento de datos los datos inicializados en el programa y a continuación se guardan los datos creados de forma dinámica por el

programa, mem_brk_point va creciendo conforme se precisa mas memoria para alojar los datos del programa. La segmento de pila empieza en la posición 0x7fffc000 y crece hacia posiciones mas bajas de la memoria. Quedando el segmento de 0x7fffc000 a 0x7fffffff desutilizado y el segmento superior a 0x7fffffff reservado para futuros desarrollos. Y es en estas posiciones superiores donde alojamos nuestra SPM. Nuestra SPM utilizara las posiciones de 0x8000000 a 0x8fffffff para alojar las instrucciones que serán ejecutadas desde esta memoria.

```
* Graphically, it all looks like this:
*
*                Virtual        Level 1    Host Memory Pages
*                Address        Page       (allocated as needed)
*                Space          Table
* 0x00000000    +-----------+    +-+      +--------------------+
*               | unused    |    | |----->| memory page (64k)  |
* 0x00400000    +-----------+    +-+      +--------------------+
*               |           |    | |
*               | text      |    +-+
*               |           |    | |
* 0x10000000    +-----------+    +-+
*               |           |    | |
*               | data seg  |    +-+      +--------------------+
*               |           |    | |----->| memory page (64k)  |
* mem_brk_point +-----------+    +-+      +--------------------+
*               |           |    | |
*               |           |    +-+
*               |           |    | |
* regs_R[29]    +-----------+    +-+
* (stack ptr)   |           |    | |
*               | stack     |    +-+
*               |           |    | |
* 0x7fffc000    +-----------+    +-+      +--------------------+
*               | unused    |    | |----->| memory page (64k)  |
* 0x7fffffff    +-----------+    +-+      +--------------------+

*/
```

Figura 21 : Subsitema de memoria del simulador.

6.3.2 - FUNCION SPM_FILL

Como hemos explicado en el apartado anterior esta es la función que debe incluirse en el código de nuestros benchmarks para poder introducir de manera sencilla las instrucciones que queremos en nuestra SPM. Esta función recibe como argumentos dos parámetros, el primero indica la posición de la primera instrucción a introducir en la SPM, y el segundo el numero de instrucciones que se van a introducir en la SPM. Las primeras tres instrucciones se encargan de sumar a la dirección pasada como primer parámetro, que estará en el registro 4, el valor 0x8000000 que sera la dirección destino que se utilizara como dirección de dicha instrucción en el segmento de memoria de la SPM. El siguiente bucle se encarga simplemente de cargar el numero de instrucciones pasada como segundo parámetro en la SPM. Para ello carga la instrucción con un load (dlw) y la guarda como un store en el rango de direcciones de la SPM (del 0x80000000 al 0x8fffffff). El código se muestra a continuación:

```
void spmfill(int addr, int instr){
    int i=0;

    __asm__ __volatile__("addu $8 ,$0 ,$0");
```

```
    __asm__ __volatile__("lui $8 ,0x8000");
    __asm__ __volatile__("or $8 ,$8 ,$4");


    for(i=0;i<(instr+4);i++){
       __asm__ __volatile__("dlw $10 ,0($4)");
       __asm__ __volatile__("dsw $10 ,0($8)");
       __asm__ __volatile__("addiu $4 ,$4 ,8");
       __asm__ __volatile__("addiu $8 ,$8 ,8");
    }
}
```

A continuación mostraremos los cambios necesarios en el código del simulador para que funcione nuestra solución.

6.3.3 - FICHERO sim_vatios.c

CAMBIOS EN ruu_fetch()

En el fetch del simulador comprobamos que si la dirección a cargar esta en la SPM, se lanza con la latencia correspondiente a la SPM:

```
static void
fetch_init(void)
{

(….)

if(fetch_regs_PC >= 0x80000000){
          //Direccion de la SPM fisica
            lat = spm_lat;
            spm_fisica_load=1;
           }

(….)
```

También añadimos código para controlar la ejecución de funciones dentro de la SPM. Se debe controlar dicha ejecución dado que las instrucciones de salto de llamada a función no cambian el ultimo bit del PC, funcionando correctamente dado que el segmento de memoria de código esta ubicado de 0x00400000 a 0x10000000. Al añadir código ejecutado en direcciones ubicadas en el segmento de memoria superior a 0x80000000 debemos controlar que cuando ejecutamos una llamada a función en la SPM, la siguiente instrucción se ejecute en la dirección adecuada, y viceversa cuando volvemos de una llamada a función. Para ello añadimos el siguiente código, donde el primer if controla que entramos en el código de la SPM, el segundo que salimos de código ejecutado en la SPM y el tercero que ejecutamos una llamada a función:

```
/* SPM FISICA FUNCIONES (SALTOS RELATIVOS A ABSOLUTOS)*/
```

```
        if(SPM_aux ==0 && fetch_regs_PC >= 0x80000000 ){
            SPM_aux = 1;
            //printf("ENTRO 0x%08x\n", fetch_regs_PC);
             function_deep[0]=funcion;
        }
        if(SPM_aux == 1 && ifq_cond == 1 && function_deep[0]==
funcion/* == 0*/ &&                fetch_regs_PC < 0x80000000){
            SPM_aux =0;
            //printf("SALGO 0x%08x\n", fetch_regs_PC );
            function_deep[0]=0;
        }
        if(SPM_aux == 1 &&(op_start_spm == JAL || op_start_spm ==
           JALR)){
            //printf("MEMORIA\n");
            fetch_regs_PC = fetch_regs_PC & 0x0FFFFFFF;

        }

/* FIN SPM FISICA FUNCIONES (SALTOS RELATIVOS A ABSOLUTOS)*/

(….)

}
```

CAMBIOS EN ruu_dispatch()

De la misma manera en el dispatch comprobamos que si ejecutamos un retorno de una función y antes de ejecutarla estábamos ejecutando código en la SPM, volvamos a ejecutar código en la SPM.

```
static void
ruu_dispatch(void)
{

(….)

           /*SPM FISICA*/
        if(SPM_aux == 1 &&(rs->op == JR) && funcion == 0 && rs
           ->IR.b == 0x1f000000){

            fetch_regs_PC = fetch_regs_PC | 0x80000000;
            fetch_pred_PC = fetch_pred_PC | 0x80000000;
            regs.regs_NPC = regs.regs_NPC| 0x80000000;

            }
        /*FIN SPM FISICA*/

(….)

}
```

CAPÍTULO 7 : SCRATCH PAD MEMORY DINAMICA

En este capítulo presentaremos una nueva arquitectura para mejorar el consumo energético y el rendimiento de una SPM. Esta propuesta consistirá en una serie de cambios en la arquitectura de la SPM, junto con una serie de nuevas instrucciones que permitirán sacarle un mayor partido a este tipo de memorias. Esta nueva arquitectura propuesta, mejora los resultados de las memorias SPM en contextos donde se requieren muchos cambios en el contenido de la SPM, ya que reduce el retraso por la actualización de código en la SPM, motivando un uso muy dinámico de esta, es decir sistemas donde se ejecutan programas con una gran cantidad de hot-spots o puntos calientes. Esta nueva arquitectura se basa en una pequeña unidad de control que carga el código ejecutado al vuelo en la SPM, mientras este es lanzado a ejecución. Permitiendo esta carga mediante una serie de nuevas instrucciones introducidas en el procesador, destinadas a tal fin.

Hay que destacar también que la técnica introducida es totalmente ortogonal permitiendo ser usada con otras técnicas existentes de elección del código a introducir en la SPM y uso eficiente de la SPM, presentadas en anteriores estudios y comentadas en el anterior capítulo.

Nuestras distintas pruebas realizadas han presentado una mejora en los resultados de una media de 30,6% en la mejora del consumo energético y de un 7,6% en el rendimiento del procesador frente a un sistema de memoria cache convencional.

Son muchos los estudios acerca de la mejora en el consumo energético y/o en el tiempo de ejecución que supone la utilización eficiente de una SPM en lugar de una cache. Todos estos estudios presentan una serie de técnicas sobre la asignación de código en la SPM que pueden dividirse en dos tipos, las que se basan en el estudio de algoritmos para mejorar el consumo energético y el rendimiento en base al contenido que debe contener la SPM en cada momento, y las técnicas que mejoran o añaden elementos hardware para una mejor gestión de los contenidos de la memoria SPM, y las técnicas mixtas que utilizan ambas opciones en este apartado comentaremos algunas de estas dos últimas técnicas.

En [1] Poletti et al. propone en su trabajo una serie de modificaciones hardware software para un sistema multitarea donde los objetos son movidos de la memoria principal a la scratchpad con la ayuda de un API creado para tal fin. Para ello propone utilizar una DMA junto con una serie de instrucciones para cargar de una manera eficiente la SPM, reduciendo gracias a la DMA el sobrecoste en las transferencias entre la memoria principal y la SPM.

Egger et al. en [7] propone una serie de cambios hardware en la MMU para el manejo de la scratchpad, permitiendo la carga en la scrathpad de los trozos de código mas frecuentemente utilizados bajo demanda. Además añade una pequeña minicache para reducir el consumo energético y mejorar el rendimiento. Mediante un algoritmo, basándose en información de profiling de los bechmarks utilizados, divide los segmentos de código en cacheables, no cacheables y páginables. Donde los dos primeros serán colocados de manera fija en la memoria principal, permitiendo al código cacheable utilizar la minicache, mientras que el páginable, segmentos de código del tamaño de una página de la MMU (Memoty Management Unit), es movido bajo demanda a la SPM. La técnica anterior solo permite la carga de código en la SPM. Basándose en esa propuesta Hyungmin Cho et al. En [9] . propone una ampliación que permite también la carga de datos en la SPM. La decisión de cargar unos datos u otros vienen determinada por la propuesta y resolución de un problema de programación dinámica entera. Egger et al. en [8] mejora la técnica presentada en [7]

mediante la utilización de un bit en cada entrada de la TLB llamado SPM flag. Este flag guardado en la TLB determina cada vez que el procesador solicita una nueva instrucción si se accede a una dirección de memoria contenida en las direcciones físicas de la SPM y si la instrucción requerida debe ser cargada de la SPM o de la minicache, ahorrándonos el acceso a una de estas dos entidades, ahorrando el tiempo y la energía necesarias para realizar una comparación de los 32 bits de la dirección requerida.

En [10] Janapsatyat et al. introduce una serie de cambios hardware-software para guardar en la spm los trozos de código mas utilizados. Se introducen una serie de instrucciones especiales en diversos puntos clave mediante un algoritmo heurístico, que activan un controlador hardware que se encarga de manejar el flujo de datos de la scratchpad. Este controlador es el encargado de para la CPU cuando se copian datos de la memoria principal a la SPM y es activado gracias a las nuevas instrucciones introducidas. Esta aproximación a diferencia de nuestra propuesta requiere conocer el tamaño de la SPM utilizada en tiempo de compilación.

Lee en [13] presenta un esquema de memoria on-chip basado en la utilización ,junto a la cache de primer nivel , de una pequeña memoria on-chip llamada loop cache cuya finalidad es reducir el consumo energético en el lanzamiento de instrucciones. Utiliza esta memoria para bucles en ella mediante una serie de instrucciones especiales que le permiten interactuar con el controlador de dicha memoria.

Nosotros en este capítulo presentamos una técnica ortogonal que puede complementarse con cualquier algoritmo de asignación de código a la spm de los propuestos por los artículos anteriores. Nuestra propuesta se basa en una serie de de pequeños cambios hardware-software para permitir la copia de instrucciones de la memoria principal a la scratchpad. Presentamos una serie de instrucciones de alto nivel que permiten mover partes del código, funciones y bucles, a la scratchpad de forma sencilla y dinámica quedando de momento en manos del programador la elección de los segmentos de código a copiar.

7.1 - SISTEMA SIMULADO

Nuestra propuesta consiste en una serie de cambios hardware-software en la arquitectura para mejorar el consumo y rendimiento de un procesador empotrado. En los siguientes apartados profundizaremos sobre los cambios propuestos, el primer apartado se centrara en los distintos cambios hardware propuestos sobre una arquitectura convencional de SPM sin un sistema DMA, mientras que el segundo apartado profundizaremos sobre las distintas instrucciones introducidas para sacarle provecho a la arquitectura propuesta.

7.1.1 - CAMBIOS HARDWARE

En esta aproximación proponemos un subsistema de memoria basado en la utilización conjunta de un buffer de instrucciones y una scratchpad memory (spm), esta configuración puede verse en la figura 22 . El buffer de instrucciones consistirá en una cache de un tamaño de una sola linea y permitirá mejorar el tiempo de acceso y el consumo energético cuando se ejecutan instrucciones de manera secuencial. Utilizaremos la scratchpad memory para ejecutar bucles y funciones que introduciremos en la spm de manera dinámica, intentando maximizar el numero de accesos a esta.

Para hacer un uso mas eficiente de la spm proponemos un sistema semi-automático de carga de instrucciones en la spm evitando la sobrecarga por la ejecución de instrucciones load store necesarias para introducir instrucciones en ella como explicaremos en el siguiente apartado.

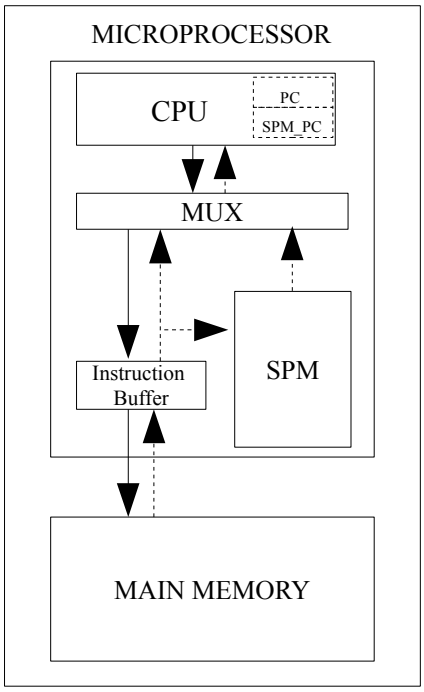

Figura 22 : Arquitectura

En nuestra solución añadimos al procesador dos modos de ejecución de instrucciones, el modo memoria y el modo scratchpad. Utilizamos el modo memoria cuando ejecutamos instrucciones desde el buffer de instrucciones, este modo no requiere ningún cambio respecto a la ejecución de instrucciones en un procesador estandard. Utilizamos el modo scratchpad cuando hagamos uso de la spm para ejecutar instrucciones, en este modo el procesador debe ser capaz de distinguir si la instrucción que se esta ejecutando esta dentro de la spm o no y en caso de que no este deberá cargar la instrucción de la memoria principal a la spm. Este proceso que explicaremos en el siguiente apartado con mas detalle queda expuesto en la figura 26. Para hacer posible este proceso hemos requerido de una serie de cambios hardware y software que a continuación detallamos. Dado que la spm esta mapeada en el conjunto de direcciones físicas del procesador proponemos añadir un segundo contador de programa que se utilizara cuando el procesador este en modo scratchpad y que se inicializará a la primera posición de la spm cuando se cambie de modo. A partir de ese momento el procesador buscara las siguientes instrucciones en la SPM y siempre que en la posicion de memoria perteneciente a la SPM requerida se encuentre una instrucción valida evitara pasar por el buffer de instrucciones y se evitara la comprobación de etiquetas, con el ahorro energético que dicha comprobación supone. Se ha añadido también una etiqueta a la spm, esta etiqueta solo se comprobara cuando se cambie de modo a fin de comprobar si el conjunto de instrucciones que se ejecutaran a continuación están o no dentro de la spm, en el caso de que la comparación de etiquetas sea correcto, se procederá a lanzar las siguientes instrucciones desde la SPM, mientras que en el caso de que sea incorrecto se limpiara la SPM arrojando en su interior instrucciones que sean consideradas invalidas, para poder cargar las instrucciones bajo demanda de la manera explicada en

el anterior punto, se cargara en la SPM el primer bloque de instrucciones requerido desde la memoria principal y se cambiará la etiqueta por la nueva etiqueta correspondiente. Para poder tener diferentes trozos de código dentro de la spm en un mismo momento, proponemos la división en bloques de la spm, esto sera especialmente útil cuando metemos funciones dentro de la spm, donde cada bloque tendrá su propia etiqueta, tal como mostramos en la figura 24. Esto nos permitirá tener en la SPM distintos "hot spots" (puntos calientes del programa, instrucciones que se repiten de manera repetida a lo largo de la ejecución del programa) al mismo tiempo ahorrándonos en muchos casos tener que volver a cargarlos, siendo especialmente útil cuando se tienen funciones dentro de bucles, ya que gracias a esta división en bloques se puede incluir el bucle dentro de un bloque y la función dentro de otro bloque, permitiendo una rápida ejecución de todo el conjunto de instrucciones que componen ese bucle. Para posibilitar esta ejecución de instrucciones dentro de bucles, se ha añadido una sencilla pila, donde guardamos el contador de programa de la SPM, antes de saltar a otro bloque dentro de la SPM. En la figura 23 se muestra un ejemplo de dicho problema.

```
_spmstart                                   _spm_start
    for(i = 0; i< 10000; i++){                  for(i = 0; i< 10000; i++){
     (....)                                          (....)
                                                     _spm_function
     foo();                                           foo();

     (....)                                          (....)

    }                                           }
_spm_end                                    _spm_end
```

Figura 23 : Comparación de código

En la figura 23 Se muestra la comparación entre dos distintos bucles utilizando las instrucciones correctas para el uso adecuado de nuestra SPM que explicaremos en el siguiente apartado. Así mientras el código de la izquierda funcionaria bien en cualquier SPM, el código de la derecha solo funcionaria correctamente en el caso de que metiéramos la función foo en un bloque distinto de la SPM y añadiéramos la pila de contadores de programa para volver a la instrucción en la SPM correcta después de ejecutar la función foo, en caso contrario la función foo y el bucle rivalizarían por las mismas posiciones en la SPM, haciendo que solo aprovechara la SPM la función foo.

7.1.2 - CAMBIOS SOFTWARE

Para interaccionar con el modelo propuesto se han creado tres nuevas instrucciones llamadas spm_start, spm_call_start y spm_end. Estas instrucciones permitirán cambiar de modo al procesador y ejecutar/cargar instrucciones o funciones en la spm. Las instrucciones spm_start y spm_call_start dispondrán de un campo inmediato donde se indicara el bloque de la spm que utilizaremos.
Las instrucciones spm_start y spm_end estan diseñadas para guardar y ejecutar bucles en la spm. Cuando se ejecute una instrucción spm_start se cambiara de modo a modo scratchpad, se inicializará el contador de programa de la spm a la primera dirección del bloque de la spm utilizado, y se comparará la dirección de la instrucción spm_start con la etiqueta de la spm, en el caso en que

la comparación sea positiva se empezara a ejecutar instrucciones de la spm, en el caso en que esta sea negativa se borrara el bloque de la spm correspondiente y se guardara como etiqueta la dirección de la instrucción spm_start. Cuando se encuentre un spm_end simplemente se procedera a cambiar de modo a modo memoria.
La instrucción spm_call_start esta especialmente diseñada para cargar/ejecutar funciones en la spm. Cuando se encuentre una instrucción de este tipo el procesador quedara a la espera de una llamada a función, cuando se ejecute esta se cambiara de modo a modo scratchpad, se inicializará el contador de programa de la spm a la primera dirección del bloque de la spm utilizado y se comparara la dirección de inicio de la función con la etiqueta del bloque de la spm correspondiente, si estas no coinciden se cambiara la etiqueta y se borrara el bloque de la spm de manera similar a la ejecución de una instrucción spm_start. El procesador volverá a modo memoria cuando encuentre la instrucción de retorno de la función dentro del rango de direcciones de la spm (dentro de la spm) sin necesidad de ejecutar un spm_end , todo este proceso puede verse de forma esquemática en la figura 25. Para ejecutar de este modo una función en la spm, cada llamada a dicha función deberá ser precedida de una instrucción spm_call_start ya que es esta instrucción la que fuerza el cambio de modo y la comparación de etiquetas. Estas instrucciones seran de tipo i o inmediato, las instrucciones de tipo inmediato soportan la inclusión de una constante de 16 bits que sera donde introduciremos el bloque a utilizar. Las instrucciones de tipo i del simulador son de 64 bits con el siguiente formato `A = 0x000000<CODE_INTR> B = <RS><RT><IMM>` donde entendemos por A los primeros 32 bits de la intrucción y B los restantes 32 bits, en los primeros bits definiremos la intrucción a utilizar y en los últimos 16 bits, el espacio reservado para el immediato el bloque de la spm a utilizar.

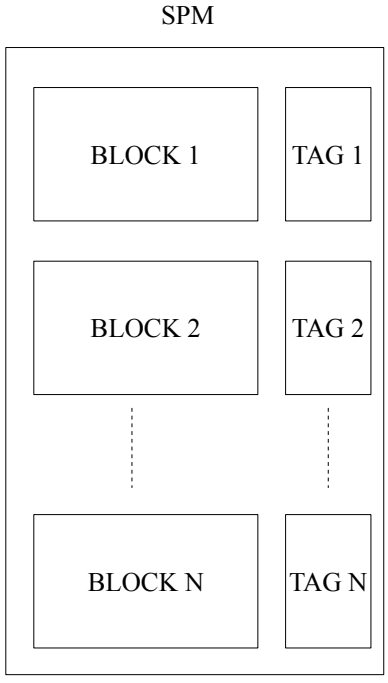

Figura 24 : Spm por bloques

Cuando el procesador esta en modo scratchpad utiliza el contador de programa de la spm, pero ambos contadores de programa se actualizan de la misma manera. Así pues cuando el programa vuelve al modo memoria no se requerirá de ningún cambio adicional. La carga de instrucciones en

la spm se realizará por petición, solo se cargaran las instrucciones que requiera la ejecución del programa. Cada vez que se lanza una instrucción dentro de la spm, se comprueba si en la dirección de la instrucción lanzada reside una instrucción valida, en el caso de que la instrucción en dicha dirección no haya una instrucción valida querrá decir que dicha instrucción no ha sido cargada aun en la spm, cargaremos entonces la linea de cache correspondiente en la spm, ayudándonos del contador de programa principal y del buffer de instrucciones, una vez cargada la linea de cache requerida en la spm se continuara ejecutando instrucciones desde la spm hasta que se cambie de modo o vuelva a aparecer otra instrucción no valida, la figura 26 ilustra este proceso de carga de instrucciones en la spm.

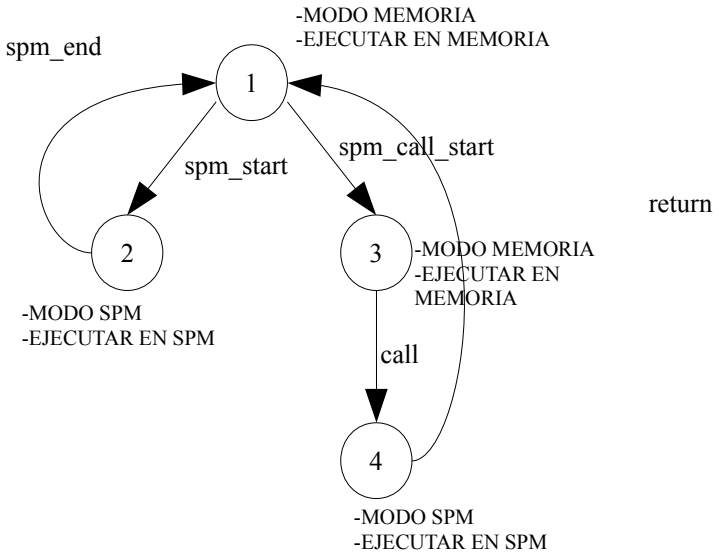

Figura 25. Cambios de modo del procesador.

7.2 - OPCIONES AÑADIDAS AL SIMULADOR

En este apartado explicaremos las distintas opciones que se han añadido al simulador para configurar la SPM deseada. Algunas de estas opciones ya fueron explicadas en el anterior capítulo.

En primer lugar definimos el tamaño en bytes del buffer de instrucciones, este buffer como hemos comentado en anteriores apartados, actuara de mini-cache permitiendo obtener un buen rendimiento en aquellas partes del código que se ejecuten secuencialmente. Por defecto si no se configura ninguna memoria cache de instrucciones de primer nivel y se incluye una memoria SPM, se crea una configuración con un buffer de 32 bytes.

```
-buffer:size   <size of temporal buffer (in bytes)>
```

También se ha incluido una opción para configurar el numero de bloques con los que cuenta la SPM. Cuantos mas bloques dispongamos más hot spots del programa podremos tener dentro de la SPM al mismo tiempo. Por defecto este valor esta a uno, es decir por defecto se actuará con SPMs de un único bloque.

```
-spm:num_blocks <block size in the scratchpad memory>
```

La siguiente opción incluida en el simulador nos permite definir el tamaño de cada uno de los bloques de la SPM. A efectos de comodidad de su uso, este tamaño viene dado por el numero de instrucciones que podrá albergar como máximo la SPM, debido a que es responsabilidad del programador no incluir en la SPM mas instrucciones de las que caben dentro de ella. Si el tamaño es 0 indicara que el sistema no cuenta con ninguna SPM en su jerarquía de memoria. El tamaño total de la SPM vendrá dado por el este valor multiplicado por 8 que es el numero de bytes que ocupa cada una de las instrucciones de 64 bits, por el numero de bloques con la que cuenta la SPM.

```
-spm:size       <size of scratchpad memory (in instructions)>
```

La siguiente opción sirve para configurar la latencia a la que actuará la SPM, es decir el número de ciclos que tarda en suministrar la SPM a la CPU una instrucción contenida dentro de la SPM. Por defecto este valor sera 1.

```
-spm:lat        <scratchpad memory latency (in cycles)>
```

Por último a efectos de depuración se ha incluido una última opción que muestra por pantalla una traza de la ejecución de las instrucciones de los bechmarks cuando usan la SPM. Esta opción simplemente sirve a efectos de depuración del simulador, estando por defecto desactivada.

```
-verbose        <operate in debug spm mode (for testing only)>
```

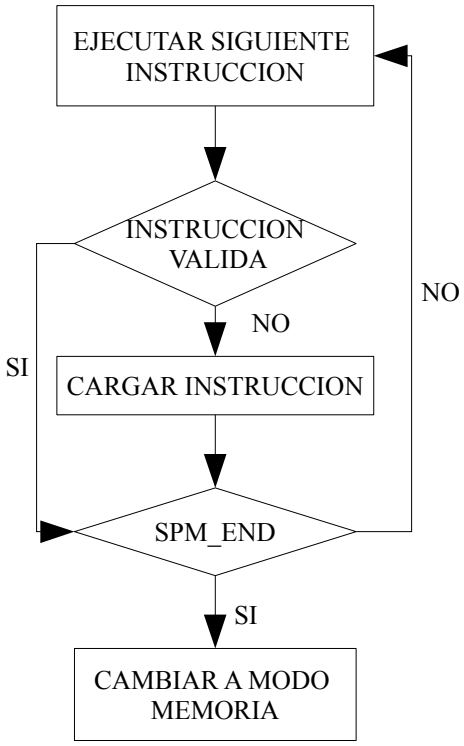

Figura 26. Carga de instrucciones en la spm

7.3 - CAMBIOS INTRODUCIDOS EN EL CÓDIGO DEL SIMULADOR

En este apartado explicaremos los cambios, más importantes , introducidos en el código del simulador para lograr la simulación del sistema con SPM comentado en los apartados anteriores. También explicaremos como introducir de manera correcta, las instrucciones explicadas en los anteriores apartados necesarias para el uso correcto de la SPM, en un lenguaje de alto nivel como C.

7.3.1 – FICHERO regs.h

En primer lugar introduciremos en el fichero regs.h, en la estructura del simulador que se encarga de guardar el estado de los contadores de programa, el contador de programa de la SPM, que se usara cuando el sistema entre el modo SPM.

```
struct regs_t {
  md_gpr_t regs_R;       /* (signed) integer register file */
  md_fpr_t regs_F;       /* floating point register file */
  md_ctrl_t regs_C;      /* control register file */
  md_addr_t regs_PC;          /* program counter */
  md_addr_t regs_NPC;         /* next-cycle program counter */
  md_addr_t regs_SPM_PC;        /* SPM program counter */
};
```

7.3.2 - FICHERO sim-vatios.c

En este fichero se concentra toda la lógica de las distintas etapas por las que pasan las instrucciones del procesador superescalar que simula el simplescalar.

<u>En sim_check_options:</u>

En esta función introducimos la lógica que valida que las distintas opciones de configuración añadidas al simulador tienen valores correctos, ademas tambien inicializamos las variables necesarias para aplicar las distintas nuevas opciones de configuración añadidas al simulador.

```
/* check simulator-specific option values */
void
sim_check_options(struct opt_odb_t *odb,        /* options
database */
            int argc, char **argv)        /* command line arguments
*/
{

(….)

/*Comprobaciones scratchpad memory*/
  if(spm_size<0)
    fatal("SPM size must be a positive number");

//Crear SPM del tamanyo indicado
```

```
  spm=(int **) malloc(sizeof(int *) * spm_blocks);

  for(i=0;i<spm_blocks;i++)
      spm[i] = (int *) calloc(spm_size/
(buffer_size/8),sizeof(int));

  if(buffer_size%8 != 0)
    fatal("buffer_size%8 must be zero");

  if(spm_size%(buffer_size/8) != 0)
    fatal("(SPM size%(buffer/8)) must be zero");

//Crear vector de etiquetas (spm)

  SPM_TAG = (md_addr_t *) calloc(spm_blocks,sizeof(md_addr_t));

//Crear vectores para el control de la spm de bloques cuando se
ejecutan funciones

  recover_SPM_PC = (md_addr_t *)
calloc(spm_blocks,sizeof(md_addr_t));
  recover_bloque =  (int *) calloc(spm_blocks,sizeof(int));
  recover_mode = (enum mode *) calloc(spm_blocks,sizeof(enum
mode));

//Vector para saber la profundidad a la que se debe ejecutar un
bloque

  function_deep = (int *) calloc(spm_blocks,sizeof(int));

  if(spm_lat < 0)
      fatal("SPM lat must be a positive number");

//Fin comprobaciones spm

(…..)
}
```

En sim_main:

Añadimos una nueva variable al simulador que controlara el numero de ciclos que se ejecutan desde la spm y desde memoria principal, esta variable se mostrara en las estadísticas finales que muestra el simulador una vez acaba la ejecución del benchmark. Esta variable se actualiza en la función sim-main, de la siguiente manera.

```
      /* go to next cycle */
      sim_cycle++;
```

```
      if((((regs.regs_SPM_PC/8)-1) < spm_size && ( processor_mode
== scratchpad || processor_mode == scratchpad_function )&&
function_deep[bloque_activo] == funcion && overflow==0) ||
(fetch_regs_PC >= 0x80000000)){
       sim_cycle_spm++;
      }
      else sim_cycle_mem++;
```

En static ruu_fetch:

Esta función se encarga de simular la etapa fetch del simulador, en ella debemos decidir de donde cogemos la instrucción a lanzar y la latencia en ciclos de procesador que tardamos en conseguirla. Esta es la función que mas cambios precisara en el simulador.

Primero cambiaremos el SPM_PC, el contador de programa de la SPM, con la misma instrucción que el contador de programa del procesador. A continuación controlaremos la carga de instrucciones desde el buffer de instrucciones. También controlaremos que no se produce un overflow en la SPM, estando en modo SPM y en caso de que se produzca se cambiara de modo y se empezaran a cargar instrucciones desde memoria principal, Debemos también controlar los accesos a funciones dentro de la SPM, para que se ejecuten las instrucciones de manera correcta desde memoria principal. Por último deberemos actualizar los accesos a SPM para que el simulador para el posterior cálculo del consumo energético.

```
static
void
ruu_fetch(void)
{
(……)

//Actualizamos SPM_PC, con la misma operacion que PC
   if( regs.regs_SPM_PC != 0x000000)
      regs.regs_SPM_PC= regs.regs_SPM_PC + (fetch_pred_PC -
anterior_PC);
   else regs.regs_SPM_PC = 0x000008;

       anterior_PC= fetch_regs_PC;
(…….)

/
************************************************************
                              SCRATCHPAD MEMORY
************************************************************/
       // Para saber si se ejecuta un spm_start dentro de
scratchpad mode

       MD_SET_OPCODE(op_start_spm, inst);
```

```
       if(verbose_mode == TRUE){
           if(op_start_spm == SPM_START){
            printf("%d 0x%08x 0x%08x 0x%08x %d\n",op_start_spm,
fetch_regs_PC,inst.a,inst.b,(inst.b & 0xffff));

           }
           if(op_start_spm == SPM_END){
            printf("LANZO UN SPM_END  0x%08x\n",fetch_regs_PC);

           }
           if(op_start_spm == SPM_CALL_START){
            printf("LANZO UN SPM_CALL_START 0x
%08x\n",fetch_regs_PC);

           }
       }


       if((processor_mode == scratchpad || processor_mode ==
scratchpad_function || overflow == 1 ) && (((regs.regs_SPM_PC/8)-
1) < (spm_size-(((SPM_BLK(buffer_size,*SPM_TAG)
+8)%buffer_size)/8))) && (ifq_cond == 1)){

           // printf("ESTOY EN  MODO SCRATCHPAD!!!! 0x%08 %d
\n",fetch_regs_PC,(regs.regs_SPM_PC/8)-1);//,
spm[(regs.regs_SPM_PC/8)-1]);

//-----------------------------------------------------------------

           /*if(function_deep[bloque_activo]==funcion)
            printf("ESTOY EN  MODO SCRATCHPAD!!!! 0x%08x %d  %d
ciclo: %d\n",fetch_regs_PC,i,(regs.regs_SPM_PC/8)-1,sim_cycle);
            */

         if(function_deep[bloque_activo]==funcion &&
spm[bloque_activo][SPM_SET(log_base2(buffer_size),(spm_size/
(buffer_size/8))-1,fetch_regs_PC)]== 0){

             //Cargar de memoria y guardar en la spm
           carga=1;

           //printf("bloque: 0x%08x
%d\n",fetch_regs_PC,SPM_SET(log_base2(buffer_size),(spm_size/
(buffer_size/8))-1,fetch_regs_PC));

           //printf("CARGO!!\n");
           spm[bloque_activo][SPM_SET(log_base2(buffer_size),
(spm_size/(buffer_size/8))-1,fetch_regs_PC)]=1;
```

```
            }
            if(overflow==1){
             overflow=0;
             processor_mode = processor_mode_overflow;
            }
        }



/*****************************************************************
                                FIN SCRATCHPAD MEMORY
*****************************************************************/

        if(i==0 && fetch_regs_PC != last_inst)//solo en superscalar
            he_cargado = 0;

//No ejecutar cache cuando esta la instruccion en la SPM

        if(spm_size>0 && (processor_mode != memory || fetch_regs_PC
>= 0x80000000)){
            if((processor_mode == scratchpad || processor_mode ==
scratchpad_function)&& carga == 0 &&
function_deep[bloque_activo]==funcion && ifq_cond == 1 &&
(((regs.regs_SPM_PC/8)-1) < (spm_size-
(((SPM_BLK(buffer_size,*SPM_TAG)+8)%buffer_size)/8)))){


             /* Wattch: add power for spm i-fetch stage */
             spm_access++;
             lat= spm_lat;


             //printf("SPM\n");
            }
            /* else{
             if(fetch_regs_PC == last_inst){
                 lat=1;//Para avanzar


    }*/
           else{//VERSION SI NO HAY SALTOS SE APROVECHA EL BUFFER
PREBUSQUEDA
             if((i==0 || he_cargado==0) && fetch_regs_PC <
0x80000000){//he cargado, carga de instruccion que no es la
primera en el fetch (superscalar)
                 //if(carga == 0)//&& function_deep[bloque_activo]
== funcion)
                 //  printf("Aqui no entro!! %d %d\n",
((regs.regs_SPM_PC/8)-1),ifq_cond);
```

```
// UTILIZAMOS BUFFER TEMPORAL PARA CARGAR EN SPM

                /* Wattch: add power for icache i-fetch stage */
                icache_access++;

                lat =
                 cache_access(cache_il1, Read,
IACOMPRESS(fetch_regs_PC),
                              NULL, ISCOMPRESS(sizeof(md_inst_t)),
sim_cycle,
                              NULL, NULL,meter_direccion);

                last_inst=fetch_regs_PC;
                carga=0;//ya has cargado la instruccion



             }
             else{
                 lat = 1;
                 carga=0;

             }
            }

            //}
            //if(fetch_regs_PC == 0x00403010)
            //  printf("%d %d funcion  %d %d overflow %d\n",
(regs.regs_SPM_PC/8)-1,(spm_size-(((SPM_BLK(buffer_size,*SPM_TAG)
+8)%buffer_size)/8)),function_deep[bloque_activo],funcion,overflow
);

            if(fetch_regs_PC >= 0x80000000){//Direccion de la SPM
fisica
             lat = spm_lat;
             spm_fisica_load=1;
            }
            if((processor_mode == scratchpad || processor_mode ==
scratchpad_function) && (((regs.regs_SPM_PC/8)-1) >= (spm_size-
(((SPM_BLK(buffer_size,*SPM_TAG)+8)%buffer_size)/8))) /*&&
overflow==0 && ifq_cond == 1*/ &&
function_deep[bloque_activo]==funcion){//Control del tamaño de
bloque
                  //overflow y cambiamos de a modo memory
             overflow=1;
             processor_mode_overflow = processor_mode;
             processor_mode= memory;
```

```
            // lat=mem_access_latency(buffer_size); //8 o 32??????
Sobra, ahora tenemos siempre una cache
            last_inst=fetch_regs_PC;

          }
       }
       /*   else{
           // address is within program text, read instruction
from memory
           lat = cache_il1_lat;
           }*/

       //Si processor_mode== memory || spm_size == 0 (poner aqui
lat=cache_il1_lat y quitar el else anterior)
       if((((processor_mode == memory ||
function_deep[bloque_activo] != funcion || ifq_cond == 0)) ||
spm_size == 0 || spm_fisica_load == 0) && fetch_regs_PC <
0x80000000 ){

           /* address is within program text, read instruction
from memory*/
           lat = cache_il1_lat;

           /* Wattch: add power for icache i-fetch stage */
           icache_access++;

       if (cache_il1)
         {

          if(cache_il1->state == lock  &&
cache_probe(cache_auxiliar,IACOMPRESS(fetch_regs_PC))!=0){
             //Esta direccion de memoria hay que meterla en cache
             meter_direccion=1;


cache_load_flush_addr(cache_auxiliar,IACOMPRESS(fetch_regs_PC));
          }



           /* access the I-cache */
           lat =
          cache_access(cache_il1, Read, IACOMPRESS(fetch_regs_PC),
                    NULL, ISCOMPRESS(sizeof(md_inst_t)), sim_cycle,
                    NULL, NULL,meter_direccion);


             //volvemos a poner la variable a 0
           meter_direccion=0;
```

```
          if (lat > cache_il1_lat){

         last_inst_missed = TRUE;

          }

        }
      //} TLB SOLO MEMORIA Y CACHE

      if (itlb)
        {
          /* access the I-TLB, NOTE: this code will initiate
          speculative TLB misses */
          tlb_lat =
         cache_access(itlb, Read, IACOMPRESS(fetch_regs_PC),
                   NULL, ISCOMPRESS(sizeof(md_inst_t)), sim_cycle,
                   NULL, NULL,0);
          if (tlb_lat > 1)
         last_inst_tmissed = TRUE;

          /* I-cache/I-TLB accesses occur in parallel */
          lat = MAX(tlb_lat, lat);
        }
      }


(…...)

 /*FUNCION DENTRO DE SCRATCHPAD */

      if(op_start_spm == JAL || op_start_spm == JALR){

          hay_funcion=1;
      }

      if(op_start_spm == JR && inst.b == 0x1f000000){

          if((funcion-1)>0)
           hay_funcion=1;
      }

      if(RUU_num < RUU_size && LSQ_num < LSQ_size)
          ifq_cond=1;


     /*FIN FUNCION DENTRO DE SCRATCHPAD*/

(…….)
```

```
}
```

En static void ruu_dispatch(void)

Esta función simula la fase dispatch del procesador, en esta fase la instrucción se decodifica por lo que podemos ya saber que clase de instrucción estamos ejecutando. Así pues si ejecutamos una instrucción spm_start o spm_call_start debemos cambiar el modo del procesador a modo spm, ejecutando las siguientes instrucciones desde el bloque requerido de la spm, y guardando en la pila las variables necesarias para un posible posterior recover o una vuelta al estado anterior del procesador cuando ejecutemos un spm_end. En el caso de ejecutar un spm_end simplemente recuperamos el estado anteior del procesador que en caso de que no se den llamadas spm_start o spm_call_start dentro de otras, será devolver el procesador al modo memory, cargando las instrucciones desde el contador de programa del procesador y lanzado las siguientes instrucciones desde el buffer de instrucciones.

```
static void
ruu_dispatch(void)
{

(…..)

       if(rs->op==SPM_END || (function_deep[bloque_activo] ==
(funcion+1) && (processor_mode == scratchpad_function || (overflow
== 1 && processor_mode_overflow == scratchpad_function)) && rs->op
== JR && rs->IR.b == 0x1f000000)){
           /* SPM_END DISPATCH
           1-Borrar el pipeline (ruu_recover())->¿?
           2-Cambiar a modo memory
           */


           //Recuperar datos de la pila
           pop(&pila_spm,&pc_spm,&bloque,&mode);

           //Guardar datos para un posible recover
           recover_SPM_PC[bloque]= regs.regs_SPM_PC;
           recover_bloque[bloque]=bloque_activo;
           recover_mode[bloque]=mode;

           //Recuperar el bloque spm anterior
           bloque_activo=bloque;


           processor_mode= mode;
           regs.regs_SPM_PC=pc_spm+regs.regs_SPM_PC ;
           if(verbose_mode == TRUE){
            if(bloque==-1){
                printf("CAMBIO A MODO MEMORIA
```

```
PRINCIPAL(DISPATCH)\n");
                processor_mode=memory;
             }
             else printf("VOY A MODO SPM BLOQUE
%d\n",bloque_activo);
           }
       }
       /*Final Instruction SPM_END*/

       /*Instruction SPM_START*/
       if(rs->op == SPM_START){
           //funcion =0;
           if(verbose_mode == TRUE)
            printf("CAMBIO A MODO SCRATCHPAD %d\n",(rs->IR.b &
0xffff));


           //Guardar SPM_PC y bloque activo para poder volver al
bloque anterior
           if(processor_mode == memory && overflow == 0)
            push(&pila_spm,0,-1,processor_mode);
           else
            if(overflow == 0)
                push(&pila_spm,
regs.regs_SPM_PC,bloque_activo,processor_mode);
            else
                push(&pila_spm,
regs.regs_SPM_PC,bloque_activo,processor_mode_overflow);

           //COMPROBAR SI EL BLOQUE ENTRA EN CONFLICTO
           if(stack_conflict(&pila_spm,rs->IR.b & 0xffff)==1)
            fprintf(stderr,"WARNING: Conflict in SPM block
%d\n",rs->IR.b & 0xffff);
           processor_mode=scratchpad;
           regs.regs_SPM_PC = 0x000000;
           bloque_activo=(rs->IR.b & 0xffff);
           function_deep[bloque_activo]=funcion;

       }
       /*Final Instruction SPM_START*/

       /*Instruction SPM_CALL_START*/

       if(rs->op == SPM_CALL_START){
           //activamos flag y guardamos bloque a utilizar
           flag_call_start = 1;
           bloque_call_start = rs->IR.b & 0xffff;
           if(verbose_mode == TRUE){
            printf("APARECE SPM_CALL_START
```

```
%d\n",bloque_call_start);
            }
        }

        if((rs->op == JAL || rs->op == JALR) && flag_call_start ==
1){


            if(verbose_mode == TRUE)
            printf("CAMBIO A MODO SCRATCHPAD FUNTION POR CALL,
BLOQUE: %d %d\n",bloque_call_start,sim_cycle);

            //Guardar SPM_PC y bloque activo para poder volver al
bloque anterior
            if(processor_mode == memory && overflow == 0)
             push(&pila_spm,0,-1,processor_mode);
            else
             if(overflow == 0)
                 push(&pila_spm,
regs.regs_SPM_PC,bloque_activo,processor_mode);
             else
                 push(&pila_spm,
regs.regs_SPM_PC,bloque_activo,processor_mode_overflow);

            //COMPROBAR SI EL BLOQUE ENTRA EN CONFLICTO
            if(stack_conflict(&pila_spm,bloque_call_start)==1)
             fprintf(stderr,"WARNING: Conflict in SPM block
%d\n",bloque_call_start);
            //cambiamos a modo funcion
            processor_mode=scratchpad_function;
            regs.regs_SPM_PC = 0x000000;
            bloque_activo=bloque_call_start;
            function_deep[bloque_activo]=funcion;
            flag_call_start=0;
            flag_call_to_commit=1;
            overflow=0;

        }

        /*Final Instruction SPM_CALL_START*/

(.....)

if((op == SPM_START) &&  SPM_TAG[(rs->IR.b & 0xffff)] != rs->PC){

            spec_mode = TRUE;
               //3
            rs->recover_inst=TRUE;
        }
```

```
        if((op == JAL || op == JALR) && flag_call_start == 1){
            if(SPM_TAG[bloque_activo]!= fetch_pred_PC){
             spec_mode = TRUE;
             //3
             rs->recover_inst=TRUE;
            }

            //Desactivamos flag
            flag_call_start=0;
        }
```

(…..)

```
 if(RUU_num >= RUU_size || LSQ_num >= LSQ_size){
     if(hay_funcion==1)
      ifq_cond=0;
     //hay_funcion=0;
     }

}
```

En ruu_recover(int branch_index)

Esta función se utiliza para simular un recover sobre el ruu del procesador, es decir es la encargada de volver a un estado anterior del procesador tras, por ejemplo una mala predicción en la ejecución de un salto condicional. En esta función insertamos código para controlar la recuperación del estado del procesador tras ejecuciones de instrucciones spm_start, spm_call_start y spm_end mal predichas. Debemos recuperar el bloque activo, el contador de programa y el modo del procesador, para esto hacemos uso de una pila definida en los ficheros stack.c y stack.h.

```
static void
ruu_recover(int branch_index)           /* index of mis-pred branch
*/
{
(…..)

/*Instruction SPM_START*/
     if(RUU[RUU_index].op == SPM_START || (RUU[RUU_index].op ==
SPM_CALL_START && flag_call_start == 0)){

            /* Salto mal predicho y despues Start_spm Acciones:
            1-Recuperar bloque activo anterior
            2-Recuperar SPM_PC anterior
            3-Cambiar de modo
         */

          pop(&pila_spm,&pc_spm,&bloque,&processor_mode);
          if(verbose_mode == TRUE){
```

```
          if(bloque==-1){
              printf("CAMBIO A MODO MEMORIA PRINCIPAL(RECOVER)\n");
          }
          else{
              printf("VOY A SCRATCHPAD(RECOVER) %d\n",bloque);
          }
         }
         regs.regs_SPM_PC= pc_spm + (fetch_regs_PC -
RUU[RUU_index].PC);


         bloque_activo= bloque;

         if(bloque==-1)
          processor_mode = memory;

     }

     /*Final Instruction SPM_START*/

      /*Instruction SPM_END*/

     if(RUU[RUU_index].op==SPM_END ||
(function_deep[bloque_activo] == funcion && (processor_mode ==
scratchpad_function || (overflow == 1 && processor_mode_overflow
== scratchpad_function)) && RUU[RUU_index]. op == JR &&
RUU[RUU_index].IR.b == 0x1f000000)){ // && processor_mode ==
memory){
       /* SPM_END RECOVER
          1-Cambiar a modo scratchpad
       */
         if(verbose_mode== TRUE)
          printf("CAMBIO A MODO SCRATCHPAD(RECOVER)
%d\n",recover_bloque[bloque_activo]);
       bloque=bloque_activo;

       if(processor_mode == memory && overflow == 0)
           push(&pila_spm,0,-1,processor_mode);
       else
       {
           if(overflow == 0)
            push(&pila_spm,((regs.regs_SPM_PC - ( fetch_regs_PC -
RUU[RUU_index].PC ))-
recover_SPM_PC[bloque]),bloque,processor_mode);
           else
            push(&pila_spm,((regs.regs_SPM_PC - ( fetch_regs_PC -
RUU[RUU_index].PC ))-
recover_SPM_PC[bloque]),bloque,processor_mode_overflow);
```

```
       //push(&pila_spm,(regs.regs_SPM_PC-(RUU[RUU_index].PC -
RUU[branch_index].PC)),bloque);//SPM_PC bajo sospecha

       }
       bloque_activo = recover_bloque[bloque];
       regs.regs_SPM_PC= recover_SPM_PC[bloque]+ (fetch_regs_PC -
RUU[RUU_index].PC);
       processor_mode=recover_mode[bloque];

      }

      /*Final Instruction SPM_END*/

(…..)
}
```

En ruu_commit(void)

Esta función simula la etapa de commit en el procesador, dado que en ella se confirma que la ejecución de la instrucción es correcta, es el momento para realizar algunas operaciones que debemos realizar cuando se ejecuta un spm_start o un spm_call_start para ejecutar las siguientes instrucciones en la spm de manera correcta. En la fase de dispatch predijimos que al ejecutar una instrucción spm_start o spm_call_start teníamos en el bloque cargadas las instrucciones correctas por ello es el momento de comprobar si esta predicción fue correcta. Así pues, en la ejecución de instrucciones spm_start y spm_call_start, si la etiqueta de la instrucción no coincide con la etiqueta guardada en la spm, borraremos el bloque de la spm requerido por la instrucción y a continuación guardaremos la nueva etiqueta de la spm, para después hacer un borrado del pipeline, necesario para empezar a cargar las instrucciones en el bloque de la spm mientras se ejecutan, y por último cambiar de modo el procesador a modo scratchpad, en caso contrario la predicción fue correcta y las siguientes instrucciones se habrán ejecutado de manera correcta.

```
static void
ruu_commit(void)
{

(…...)

 /*Instruction SPM_START*/
      if(rs->op == SPM_START){

       if(SPM_TAG[(rs->IR.b & 0xffff)] != rs->PC){
           /* Posibilidad 1: Etiqueta != PC
           1-Borrar SPM
           2-Cambiar etiqueta
           3-Borrar el pipeline (ruu_recover())
           4-Cambiar de modo
           5-Inicializar SMP_PC
           6-Cambiar bloque activo
```

```
        */
           if(verbose_mode == TRUE)
            printf("CAMBIO A MODO SCRATCHPAD(COMMIT) %d\n",(rs-
>IR.b & 0xffff));

           spm_delete((rs->IR.b & 0xffff));
           SPM_TAG[(rs->IR.b & 0xffff)]=rs->PC;
           ruu_recover(rs -RUU);
           tracer_recover();
           bloque_activo=(rs->IR.b & 0xffff);

           ruu_fetch_issue_delay = ruu_branch_penalty;

           fetch_pred_PC=rs->PC+8;
           processor_mode=scratchpad;
           regs.regs_SPM_PC = 0x000000;
           committed== ruu_commit_width;
           //break;
       }


      }
     /*Final Instruction SPM_START*/

      /*Instruction SPM_CALL_START*/
      if((rs->op == JAL || rs->op == JALR) &&
function_deep[bloque_activo] == funcion && processor_mode ==
scratchpad_function){
       if(SPM_TAG[bloque_activo] != rs->next_PC &&
flag_call_to_commit == 1){
           /* Posibilidad 1: Etiqueta != PC
           1-Borrar SPM
           2-Cambiar etiqueta
           3-Borrar el pipeline (ruu_recover())
           4-Cambiar de modo
           5-Inicializar SMP_PC
           6-Cambiar bloque activo
        */
        if(verbose_mode == TRUE)
        printf("CAMBIO A MODO SCRATCHPAD FUNCTION(COMMIT) %d 0x
%08x 0x%08x %d\n",bloque_activo,rs->PC,rs->next_PC,sim_cycle);
           //funcion=0;
           spm_delete(bloque_activo);
//MAL TAG DE LA DIRECCION DESTINO DEL SALTO NO ES ESA
           SPM_TAG[bloque_activo]=rs->next_PC;
           ruu_recover(rs -RUU);
           tracer_recover();
```

```
        /* stall fetch until I-fetch and I-decode recover */
        ruu_fetch_issue_delay = ruu_branch_penalty;
        /*FIN AÑADIDO*/

        fetch_pred_PC=rs->next_PC;
            regs.regs_SPM_PC = 0x000000;
            committed== ruu_commit_width;

        }
        flag_call_to_commit=0;
    }
       /*Final Instruction SPM_CALL_START*/

(…...)

}
```

7.3.3 – FICHERO stack.c

En este fichero definimos la pila que utilizamos para recuperar el estado anterior del procesador, después de un recover o tras ejecutar un spm_end. En esta pila se guarda el anterior modo del procesador, el contador de programa de la spm y el bloque de la spm que se esta utilizando actualmente, solo en el caso de que se utilicen distintos bloques de la spm uno dentro de otro, como explicamos en el anterior apartado.

```
#include "stack.h"
#include "machine.h"
#include <stdio.h>
#include <stdlib.h>

void push(t_nodo **primero,md_addr_t PC,int block,enum mode modo)
{
   t_nodo *nuevo;

   //Creamos un nuevo nodo
   nuevo = (t_nodo*)malloc(sizeof(t_nodo));

   //Insertamos los datos
   nuevo->SPM_PC = PC;
   nuevo->bloque=block;
   nuevo->processor = modo;

   //Metemos en primer lugar
   nuevo->siguiente = *primero;
   *primero = nuevo;
}

void pop(t_nodo **primero, md_addr_t *PC,int *bloque,enum mode
*modo)
```

```
{
   t_nodo *aux;      // Variable auxiliar para manipular el nodo

   aux = *primero;
   if(aux == NULL){   // Si no hay elementos en la pila damos un
error
       fprintf(stderr,"Error pop sobre pila vacia!!\n");
       return;
   }
   //Reordenamos la pila
   *primero = aux->siguiente; // La pila empezará ahora a partir
del siguiente elemento

  //Devolvemos los datos requeridos por argumento
   *PC=aux->SPM_PC;
   *bloque=aux->bloque;
   *modo = aux->processor;

  //Eliminamos el elemento
   free(aux);

}

//Comprueba que el bloque dado por parametro no esta siendo
utilizado.
int stack_conflict(t_nodo **primero,int bloque){
    t_nodo *aux;      // Variable auxiliar para manipular el nodo

    for(aux = *primero;aux != NULL; aux = aux ->siguiente){
     if(aux->bloque == bloque)
         return 1;//Conflito de bloque
    }
    return 0; //No se encontro ningun conflicto
}
```

7.3.4 – FICHERO pisa.def

En este fichero se definirán las nuevas instrucciones añadidas al procesador así como su comportamiento. Este fichero es usado por la función ruu_ dispatch para ejecutar una u otra instrucción.

```
(.…)

#define SPM_START_IMPL                                    \
  {                                              \
      /* Lo que hay que hacer*/                                \
      if (SPM_TAG == regs.regs_PC);                              \
      /*    SET_GPR(RD, 1);        */            \
      /*else                       */            \
```

```
      /*SET_GPR(RD, 0);                                */
\
   }
DEFINST(SPM_START,          0xd3,
     "spm_start",         "i",
     IntALU,         F_ICOMP|F_IMM,
     DNA, DNA,       DNA, DNA, DNA)

#define SPM_END_IMPL                                        \
  {                                                 \
      /* NADA */                                            \
   }
DEFINST(SPM_END,              0xd4,
     "spm_end",               "",
     NA,                 F_ICOMP,
     DNA, DNA,       DNA, DNA, DNA)

#define SPM_CALL_START_IMPL                                 \
  {                                                 \
      /* NADA */                                            \
   }
DEFINST(SPM_CALL_START,            0xd5,
     "spm_call_start",        "i",
     IntALU,             F_ICOMP|F_IMM,
     DNA, DNA,       DNA, DNA, DNA)

(….)

#undef SPM_START_IMPL
#undef SPM_END_IMPL
#undef SPM_CALL_START_IMPL
```

7.3.5 - FICHERO access_vatios.c

Este fichero es una nueva incorporación del simulador vatios al simplescalar para el cálculo del consumo energético del simulador. En este fichero se inicializan las variables para el cálculo del consumo energético y ademas se crea un fichero con los distintos acceso a los distintos componentes del sistema que se podrá utilizar separadamente para calcular el consumo energético, parametrizando distintas opciones de los componentes utilizados, sin tener que volver a ejecutar otra vez la simulación del bechmark. En nuestro caso utilizaremos este fichero para inicializar las distintas variables de acceso de la SPM para poder calcular de manera correcta el consumo energético de esta.

```
(….)
counter_t spm_access;
(….)
static counter_t total_spm_access=0;
(….)
static counter_t max_spm_access=0;
```

```
(….)
void clear_access_stats()
/* Once per cycle.*/
{
(….)
spm_access=0;

(….)
}

void     update_access_stats(access_data_t     *     access_data)
{
(….)

  total_spm_access+=spm_access;

(….)

  max_spm_access=MAX(spm_access,max_spm_access);

(….)

access_data->accesses[SPM][spm_access]++;

(….)

}
```

7.3.6 - FICHERO power.c

Este es el fichero que contiene el código que se encarga del cálculo del consumo energético. La función que se encarga del cálculo es la función calculate que recibe un vector con los accesos al componente del procesador que se desea calcular, proporcionado por access_vatios.c y utilizando la información de cálculo proporcionada por power.h llama a CalculatePowerAndDisplay que se encarga del cálculo y de mostrar los resultado por pantalla.

```
void calculate(     int accesses[][MAX_ACCESS_CYCLE],
                    double activity[][MAX_ACCESS_CYCLE],
                    int energyModels[NUM_UNITS],
                    double userPowerAFIndependent[NUM_UNITS],
                    double userPowerAFDependent[NUM_AFS]){
(….)

     //Because CACTI shouldn't write anything.
     FILE * backUp;
     backUp = stdout;
     stdout = fopen("/dev/null","w");
     #include "calculatePower.h"
     stdout = backUp;
```

```
(….)


calculateEnergyAndDisplay(accesses,SPM,power,1,FALSE,NULL,0,totalE
nergy);

(….)

}
```

7.3.7 - FICHERO calculatePower.h

Este fichero es el utilizado para parametrizar el cálculo de los distintos componentes del procesador que se desea calcular su consumo, es el fichero que se utiliza en el la función calculate de power.c. En este fichero incluimos el código para calcular el consumo energético de la spm, para ello utilizaremos la herramienta CACTI []. CACTI es una herramienta para modelar memorias on-chip que incluye un modelo para calcular el área en silicio y el consumo energético de la memoria. El simulador calcula tres valores calculados según la técnica de Clock Gating que se aplique al procesador. El Clock Gating es una técnica que inhibe la señal de reloj a determinadas zonas de un circuito para que no se produzca conmutación los ciclos que esa zona esta inactiva. Los 3 estilos de Clock Gating que calculamos son los siguientes:

NCC: No aplicar Clock Gating
CC1: Clock Gating simple, una unidad bien esta inactiva, con un consumo nulo, o bien su consumo es el 100% del consumo máximo.
CC2: Clock Gating lineal ideal, en el que el consumo de una unidad es lineal en función de los recursos utilizados durante cada ciclo (por ejemplo, número de puertos en un banco de registros).
El código escrito para el cálculo del consumo de la spm es el siguiente:

```
        /******************** SPM ***************************/

double
spm_decoder,spm_wordline,spm_bitline,spm_senseamp,spm_power;

cache=1;


if(spm_size > 0){
c = spm_size*spm_blocks*8; /* C */
b = 8;//spm_size*8; /* B *///tamaño de cada bloque, no numero de
bloques  cache_il1->bsize
a = 1; /* A */

result2 = cacti_interface(c,b,a,0/* RW ports*/, 1, 1, 0,1/*Number
of banks*/,((double)technologyPoint)/1000, 64, 0, 0, 0, 0);
```

```
ndwl=result2.result.best_Ndwl;
ndbl=result2.result.best_Ndbl;
nspd=result2.result.best_Nspd;

if (nspd==0){nspd =1;}

rowsb = c/(b*a*ndbl*nspd);
colsb = 8*b*a*nspd/ndwl;

predeclength = rowsb * (RegCellHeight + WordlineSpacing);
wordlinelength = colsb *  (RegCellWidth + BitlineSpacing);
bitlinelength = rowsb * (RegCellHeight + WordlineSpacing);

spm_decoder =
ndwl*ndbl*array_decoder_power(rowsb,colsb,predeclength,1,1,cache);
spm_wordline =
ndwl*ndbl*array_wordline_power(rowsb,colsb,wordlinelength,1,1,cach
e);
spm_bitline =
ndwl*ndbl*array_bitline_power(rowsb,colsb,bitlinelength,1,1,cache)
;
spm_senseamp = ndwl*ndbl*senseamp_power(colsb);

spm_power = spm_decoder + spm_wordline + spm_bitline +
spm_senseamp;

}
else{
    energyModels[SPM]=0;
}



//Power model selection
       switch(energyModels[SPM]){
        case 0: /*Don't count this unit*/
            power.powerAFIndependent[SPM] = 0.0;
            break;
        case 1: /*Default power model*/
            power.powerAFIndependent[SPM] = spm_power;
            power.powerAFIndependent[SPM] *= crossover_scaling;
            break;
        case 2:
            power.powerAFIndependent[SPM] =
userPowerAFIndependent[SPM];
            break;
        default:
            fprintf(stderr,"Invalid energyModel for %s\n",
                 varUnitNames[SPM]);
```

```
            exit(-1);
        }
```

7.3.8 - Uso en cualquier bechmark en C

La utilización de las instrucciones comentadas en anteriores apartados para hacer uso de la SPM, en cualquier benchmark es bastante sencilla. Simplemente hay que introducir en el momento deseado las instrucciones en el código, para eso hacemos uso de la función asm que en C nos permite introducir instrucciones en ensamblador en nuestro código de alto nivel. Para facilitar su uso nos declararemos las instrucciones mediante la directiva #define que nos permite definir un macro para ser utilizado de forma sencilla en todo nuestro código. Las instrucciones del simulador son de 64 bits con el siguiente formato `A = 0x000000<CODE_INTR> B = <RS><RT><IMM>` donde entendemos por A los primeros 32 bits de la instrucción y B los restantes 32 bits, en los primeros bits definiremos la instrucción a utilizar y en los últimos en el espacio reservado para el inmediato el bloque de la spm a utilizar . A continuación mostramos un sencillo ejemplo:

```
/*SPM*/
//SPM_START POR BLOQUES
#define __spm_start_0                                    \
    ({ asm(".long (0x000000d3)");asm(".long (0x00000000)"); })
#define __spm_start_1                                    \
    ({ asm(".long (0x000000d3)");asm(".long (0x00000001)"); })
#define __spm_start_2                                    \
    ({ asm(".long (0x000000d3)");asm(".long (0x00000002)"); })
#define __spm_start_3                                    \
    ({ asm(".long (0x000000d3)");asm(".long (0x00000003)"); })

//SPM_END
#define __spm_end                                        \
    ({ asm(".long (0x000000d4)");asm(".long (0x00000000)"); })

//SPM_CALL POR BLOQUES

//FORMATO DE LA INSTRUCCION: A = 0x000000<CODE_INTR> B =
<RS><RT><IMM>

#define __spm_call_start_0                                \
    ({ asm(".long (0x000000d5)");asm(".long (0x00000000)"); })
#define __spm_call_start_1                               \
    ({ asm(".long (0x000000d5)");asm(".long (0x00000001)"); })
#define __spm_call_start_2                                \
    ({ asm(".long (0x000000d5)");asm(".long (0x00000002)"); })
#define __spm_call_start_3                                \
    ({ asm(".long (0x000000d5)");asm(".long (0x00000003)"); })
#define __spm_call_start_4                                \
    ({ asm(".long (0x000000d5)");asm(".long (0x00000004)"); })
```

```
void compress(void)
{
  register long fcode;
  register code_int i = 0;
  register int c;
  register code_int ent;
  register int disp;
  register code_int hsize_reg;
  register int hshift;


  offset = 0;
  bytes_out = 3;                /* includes 3-byte header mojo */
  out_count = 0;
  clear_flg = 0;
  ratio = 0;
  in_count = 1;
  checkpoint = CHECK_GAP;
  maxcode = MAXCODE(n_bits = INIT_BITS);
  free_ent = ((block_compress) ? (FIRST) : (256) );

  ent = getbyte ();

  hshift = 0;

  __spm_start_0;
  for ( fcode = (long) hsize;  fcode < 65536L; fcode *= 2L )
  {
    hshift++;
  }
  __spm_end;

  hshift = 8 - hshift;          /* set hash code range bound */

  hsize_reg = hsize;
  cl_hash( (count_int) hsize_reg);          /* clear hash table */

  __spm_start_1;
  while ( InCnt > 0 )          /* apsim_loop 11 0 */
  {
    int apsim_bound111 = 0;

    c = getbyte();   // decrements InCnt

    in_count++;
    fcode = (long) (((long) c << maxbits) + ent);
    i = ((c << hshift) ^ ent);    /* xor hashing */
```

```
    if ( htabof (i) == fcode ) {
      ent = codetabof (i);
      continue;
    } else if ( (long)htabof (i) < 0 ) { /* empty slot */
      goto nomatch;
    }


    disp = hsize_reg - i;                /* secondary hash (after G. Knott) */
    if ( i == 0 ) {
      disp = 1;
    }

probe:

    if ( (i -= disp) < 0 ) {  /* apsim_loop 111 11 */
      i += hsize_reg;
    }

    if ( htabof (i) == fcode ) {
      ent = codetabof (i);
      continue;
    }

    if ( (long)htabof (i) > 0 && (++apsim_bound111 < in_count) )
      goto probe;
nomatch:

    out_count++;
    ent = c;
    if ( free_ent < maxmaxcode ) {
      codetabof (i) = free_ent++;          /* apsim_unknown codetab */
      htabof (i) = fcode;                  /* apsim_unknown htab */
    } else if ( ((count_int)in_count >= checkpoint) && (block_compress) ) {

        __spm_call_start_0;
      cl_block ();
    }

  }
  __spm_end;

  if(bytes_out > in_count) { /* exit(2) if no savings */
    exit_stat = 2;
  }
  return;
}
```

CAPÍTULO 8 : SIMULACIONES Y RESULTADOS

En este capitulo presentaremos los resultandos obtenidos de distintas simulaciones sobre los distintos sistemas de memorias on-chip presentados en los anteriores capítulos. Así mismo explicaremos las configuraciones utilizadas, los bechmarks usados para hacer las pruebas sobre estos sistemas y una serie de comparaciones entre distintos sistemas de memorias on-chip. En primer lugar compararemos una cache convencional con la SPM diseñada en el anterior capítulo de esta memoria.

Para comparar la cache con la spm, nosotros hemos utilizado el simulador Vatios [14]. Vatios es un simulador basado en el popular simulador SimpleScalar [15] que de manera semejante al simulador Wattch [16] añade un modelo para calcular el consumo energético de ambas entidades. Para esto necesitamos añadir al simulador un modelo acorde al consumo energético de ambas memorias. Una comparacion de los valores obtenidos, en concepto de energía por acceso para ambas memorias en chip puede verse en la Tabla 1.

Tabla 1: energia por acceso.

Memory size	Cache	Scratchpad	Ratio
256B	0,55 W	0,095 W	5,78
512B	0,59 W	0,1174 W	5,02
1024B	0,65 W	0,20 W	3,25

Vatios presenta una serie de ventajas en el calculo de la potencia y consumo energético respecto a Wattch que hacen que el resultado obtenido se aproxime mas al resultado real . Para calcular el consumo energético de la spm y de la cache Vatios se basa en el modelo de CACTI [17], las principales ventajas en el calculo energético de la spm respecto a la cache se basan en la casi total ausencia de etiquetas en la spm en comparación con la cache, la mejora puede verse la tabla anterior . Para que pueda haber una comparación justa entre ambas se ha simulado utilizando en ambas memorias la misma tecnología de fabricación.

Los benchmarks utilizados para la comparación de ambas memorias onchip han sido sacados de The Mälardalen WCET research group [18], son bechmarks especialmente escogidos para que el calculo del WCET sea sencillo de realizar y han sido cuidadosamente modificados por nosotros para incluir las instrucciones precisas para la correcta utilización de la spm.

El flujo del experimento es el que mostramos en la figura 27. Empezamos con el programa escrito en código C, a continuación añadimos las nuestras instrucciones al programa para la configuración de la scratchpad. Compilamos el código con el simulador sslittle-gcc para obtener el código maquina. Este código maquina generado es simulado utilizando el simulador sim-vatios con el que obtendremos una traza de la ejecución que utilizara la herramienta power-vatios para calcular el consumo energético.

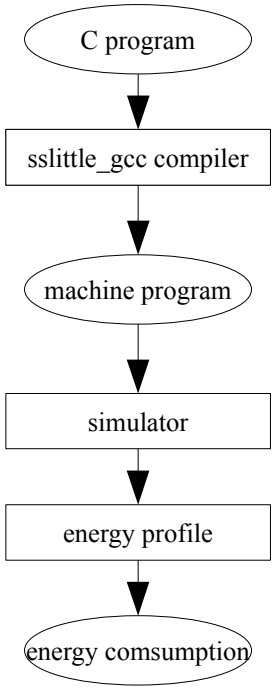

Figura 27. Flujo de trabajo

Para calcular nuestros resultados hemos seleccionado una serie de bechmarks , en concreto siete, sacados de The Mälardalen WCET research group [18] como hemos explicado anteriormente. Los ocho benchmarks estudiados son *: "bsort100" a bubblesort program , "cnt" cuenta los números no negativos en una matriz, "compress" programa de compresión de datos, "cover" testea múltiples rutas, "expint" programa que interactua con series expansion for computing an exponential integral function, "fdct" programa que calcula la transformada discreta del coseno(fast discrete cosine transform), y "fir" algoritmos de procesamientos de señales sobre filtros de respuesta finita al impulso sobre una muestra de 700 elementos.. Sobre estos benchmarks hemos realizado pruebas de rendimiento y consumo energético utilizando tres tamaños diferentes de memorias on-chip, 128B, 256B y 512B. En el caso de la cache hemos utilizado arquitecturas de cache de correspondencia directa mientras que en la spm hemos optado por una arquitectura de un único bloque debido a que los benchmarks estudiados carecen de puntos calientes concurrentes, por lo que su división en bloques afectaría al tamaño de cada bloque empeorando los resultados. Los resultados obtenidos pueden observarse en las gráficas presentadas (figuras 28 29 30 31 32 y 33).

Podemos observar que para tamaños de 128B (figuras 28 y 29) la spm se comporta mejor tanto en rendimiento como en consumo energético en todos los benchmarks. La spm presenta una mejora media de un 17% en rendimiento , en parte gracias a nuestro buffer de instrucciones, y de un 29% en consumo energético respecto a la cache. Donde los mejores resultados obtenidos se muestran en el benchmark fir en el cual obtenemos una mejora en rendimiento del 44% y del consumo energético del 53%.

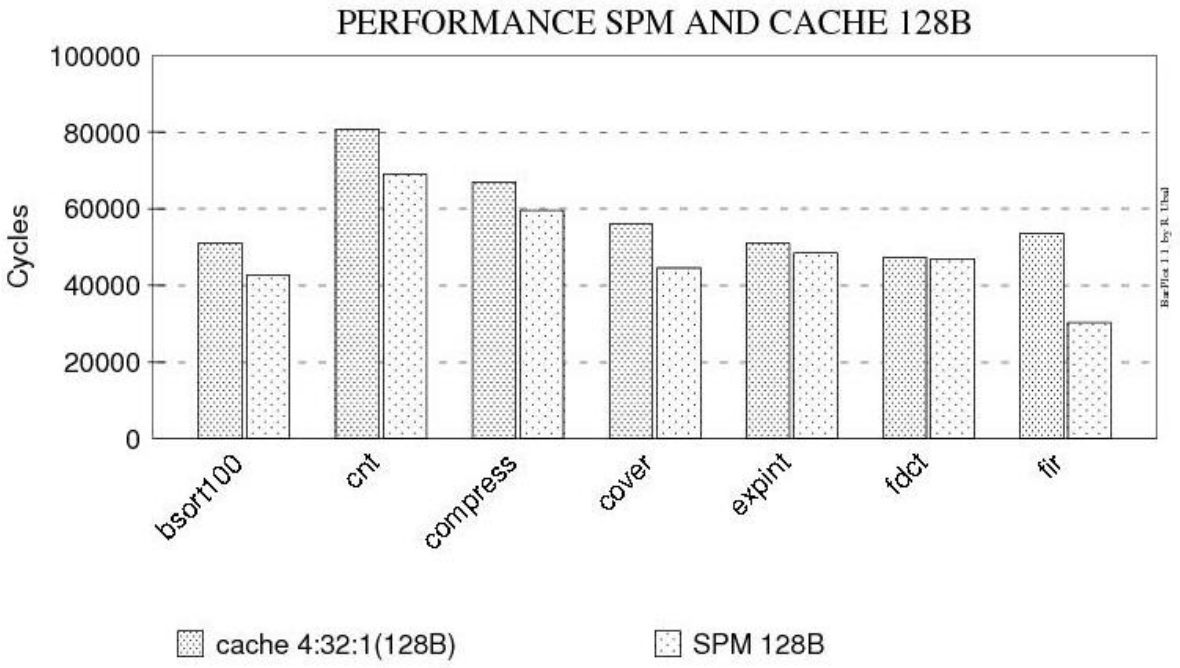

Figura 28 : Comparación de rendimiento entre cache y SPM de 128B.

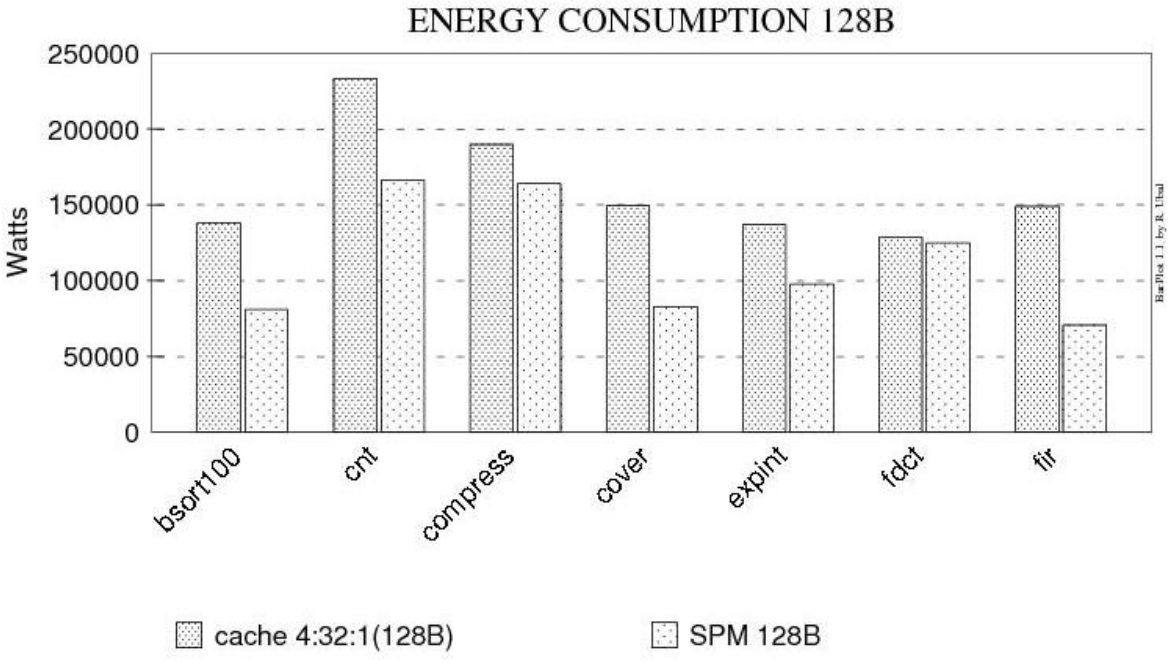

Figura 29 : Comparación de consumo entre cache y SPM de 128B.

Para tamaños de 256B (figuras 30 y 31) de los ocho benchmarks utilizados solo expint obtiene

menos rendimiento en la spm que en la cache . Este programa consta de dos bucles anidados donde el bucle exterior no cabe entero dentro de la spm y el bucle mas anidado solo se ejecuta bajo determinadas circunstancias, lo que hace que en cualquier caso la cache tome cierta ventaja con respecto a nuestra solución con spm para este tamaño. En global para 256B la spm presenta una mejora del 9% en rendimiento y del 31% en consumo energético con respecto a la cache.

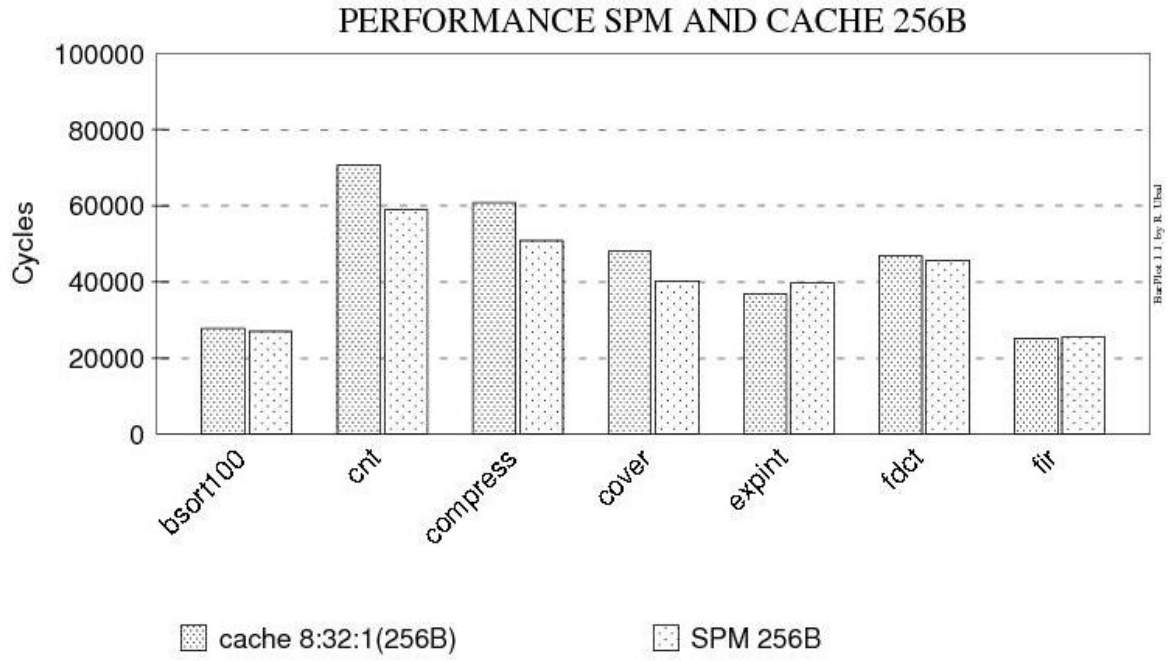

Figura 30 : Comparación de rendimiento entre cache y SPM de 256B.

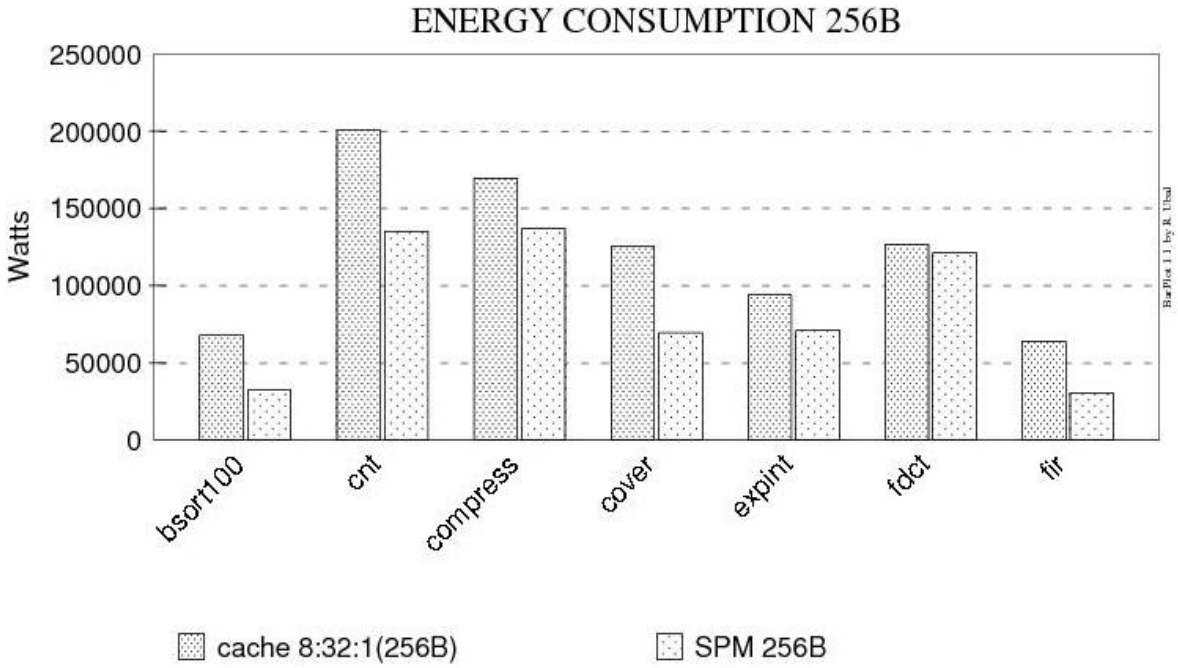

Figura 31 : Comparación de consumo entre cache y SPM de 256B.

Mientras que comparando la spm con la cache para tamaños de 512B (figuras 32 y 33) observamos que con respecto al rendimiento ambas entidades se comportan de manera semejante,aunque la cache presenta mejores resultados en cinco de los ocho benchmarks estas diferencias no son significativas presentando de media un empeoramiento del 3% del rendimiento de la spm con respecto a la cache. Sin embargo debido a su menor consumo la spm sale victoriosa en la comparación del consumo energético en todos los bechmarks, presentando una mejora media del 32% con respecto a la cache.

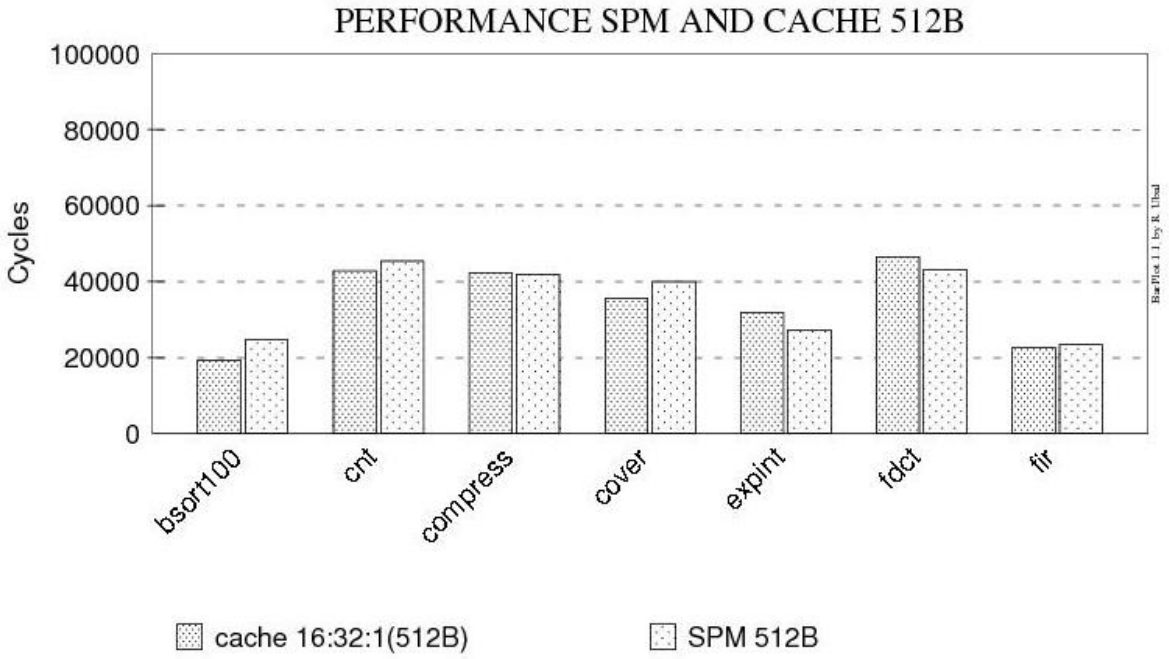

Figura 32 : Comparación de rendimiento entre cache y SPM de 512B.

Podemos concluir que el mejor aprovechamiento de la memoria on-chip y el menor consumo del acceso de la SPM con respecto a una cache convencional hacen que en general podemos observar una mejora media de la spm en comparación a la cache del 7,6% con respecto al rendimiento y del 30,6% respecto al consumo energético, que en estos sencillos bechmarks toman más importancia cuando tratamos tamaños reducidos de memorias onchip.

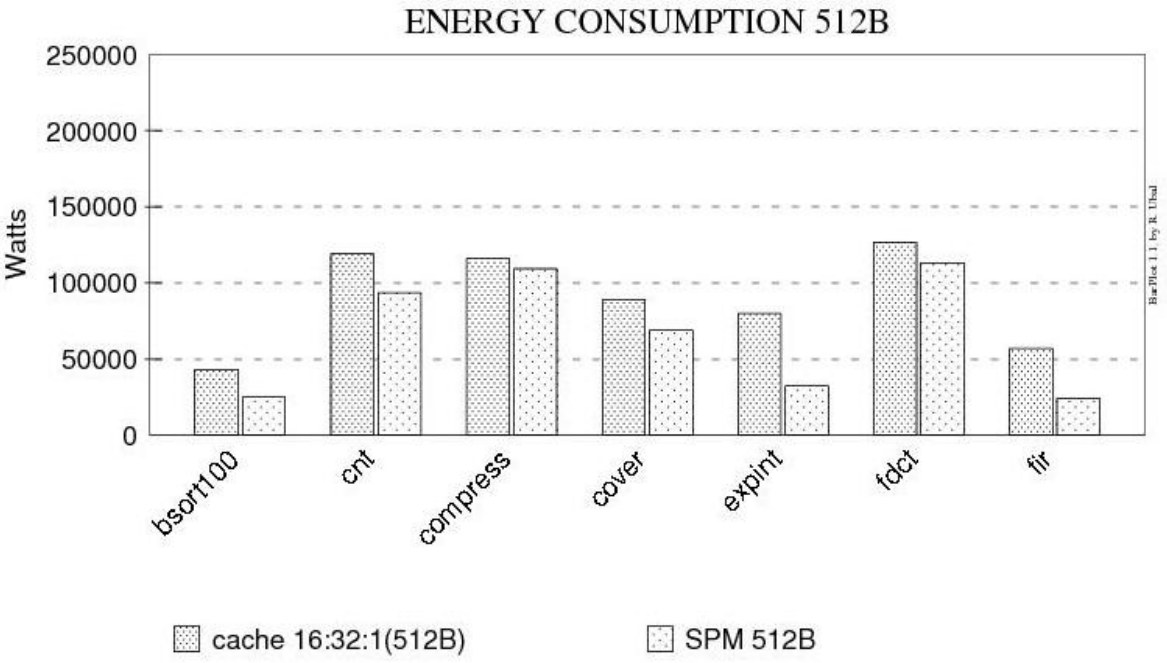

Figura 33 : Comparación de consumo entre cache y SPM de 512B.

A continuación presentamos una comparación entre las dos SPMs presentadas en esta memoria. La SPM presentada en el último apartado presenta importantes mejoras en el rendimiento y en el consumo energético en la carga de instrucciones en la SPM, lo que se hace notar en sistemas que presentan aplicaciones con muchos puntos calientes o hotspots en su código y que deben cambiar el contenido de la SPM, en multitud de ocasiones. Así pues, utilizando una SPM de 8KB, lo suficientemente grande para guardar en ella todas las instrucciones del hotspot mas grande de entre todos los programas utilizados, hemos obtenido los resultados mostrados en las figuras 34 y 35. Donde podemos observar una mejora media del 45.95% en rendimiento y del 26.36% en consumo energético. Siendo significativo observar que la nueva SPM obtenida mejora en todos los benchmarks tanto en consumo energético con en rendimiento a la SPM convencional donde las cargas de instrucciones se realizan mediante instrucciones de load/store. Significativos resultan los resultados tanto del bechmark cover, donde la mejora en rendimiento se aproxima al 84% y la del consumo al 90% debido a que la naturaleza del programa resulta especialmente beneficiosa para el nuevo sistema de SPM elaborado en esta memoria, con grandes cantidades de instrucciones que cargar en la SPM y hotspots que se solapan y se ejecutan no en demasiadas ocasiones, como el caso del benchmark adpcm donde la mejora se traduce en un 2,88% en rendimiento y un 2,66% en consumo energético debido a la gran cantidad de veces que se ejecutan los pocos hotspots que contiene el programa lo que reduce la ventaja de nuestra SPM solo a la carga inicial de estos.

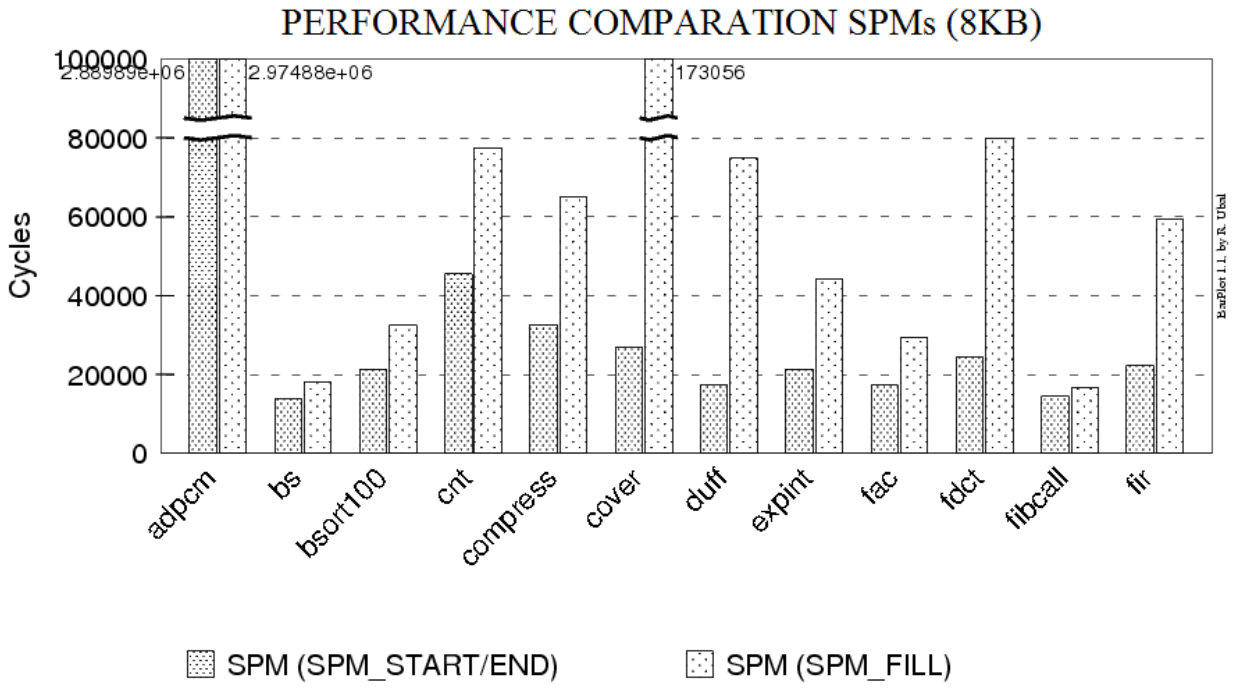

Figura 34 :Comparación de rendimiento entre SPMs de 8KB

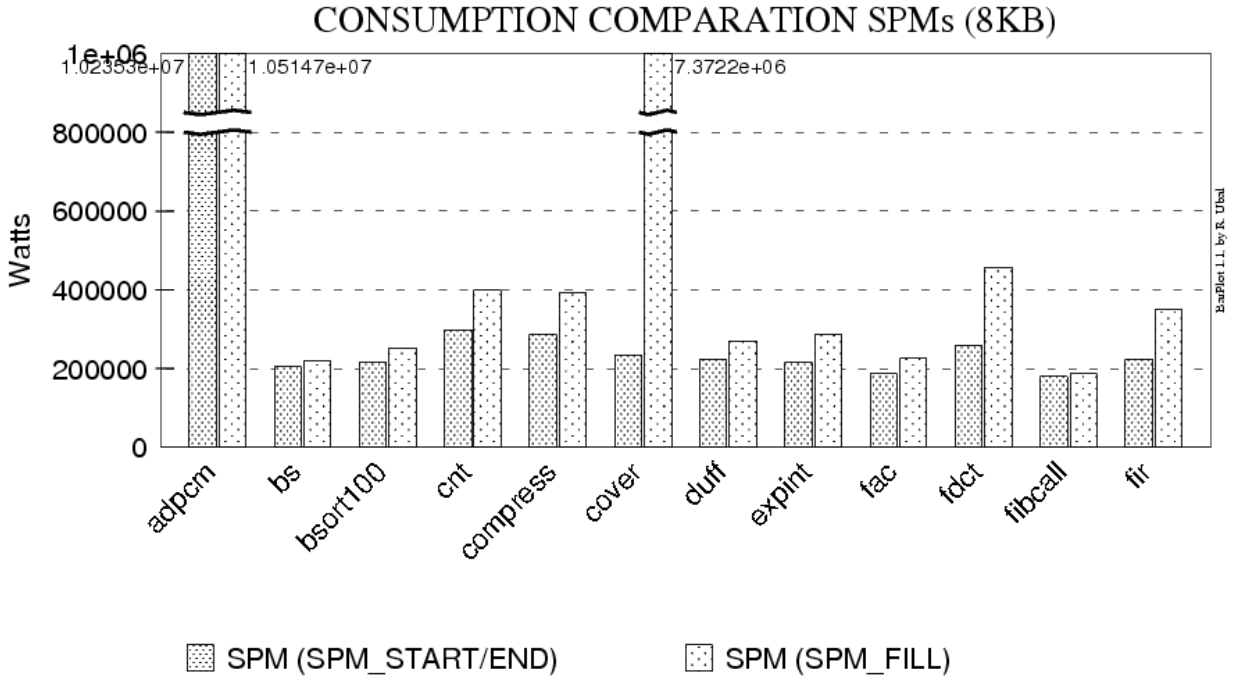

Figura 35 : Comparación de consumo entre SPMs de 8KBs.

Por último utilizamos los bechmarks para hacer una comprobación sobre la segmentación en varios bloques de la SPM. Debido a la poca complejidad de los bechmarks utilizados, al ser programas sencillos utilizados para sistemas que ejecutan aplicaciones que tienen restricciones de tiempo real,

la ventaja de utilizar varios bloques queda en entredicho al perder espacio de memoria en las bloques y ser utilizados ambos de una manera concurrente en un mismo bechmark en muy pocas ocasiones. Así pues las figuras 36 y 37 muestran el resultado de la ejecución de los siete benchmarks descritos anteriormente con dos SPM de 128B, una con un único bloque de 128B y la otra con dos bloques de 64B. La SPM con un único bloque obtienen por lo antes comentado una mejora media del 13.34% en rendimiento y del 18.44% en consumo energético.

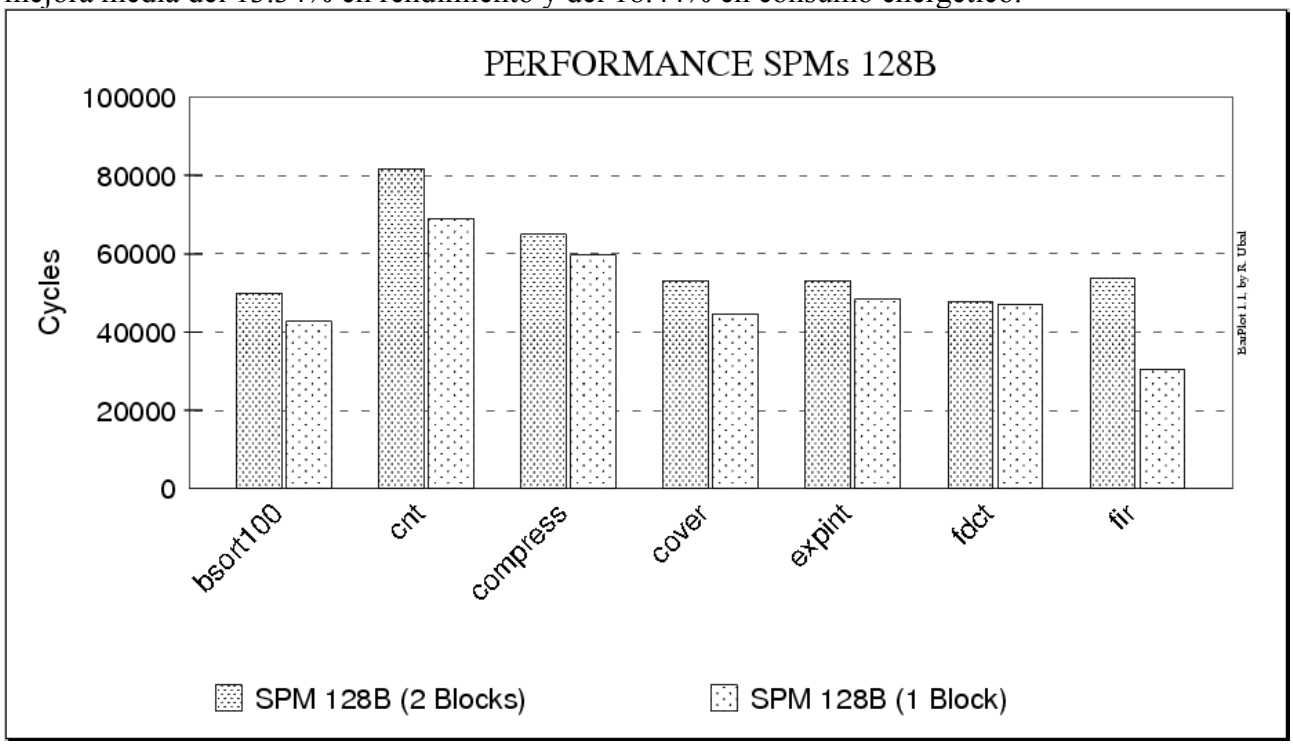

Figura 36 : Comparación de rendimiento entre SPMs de 128B.

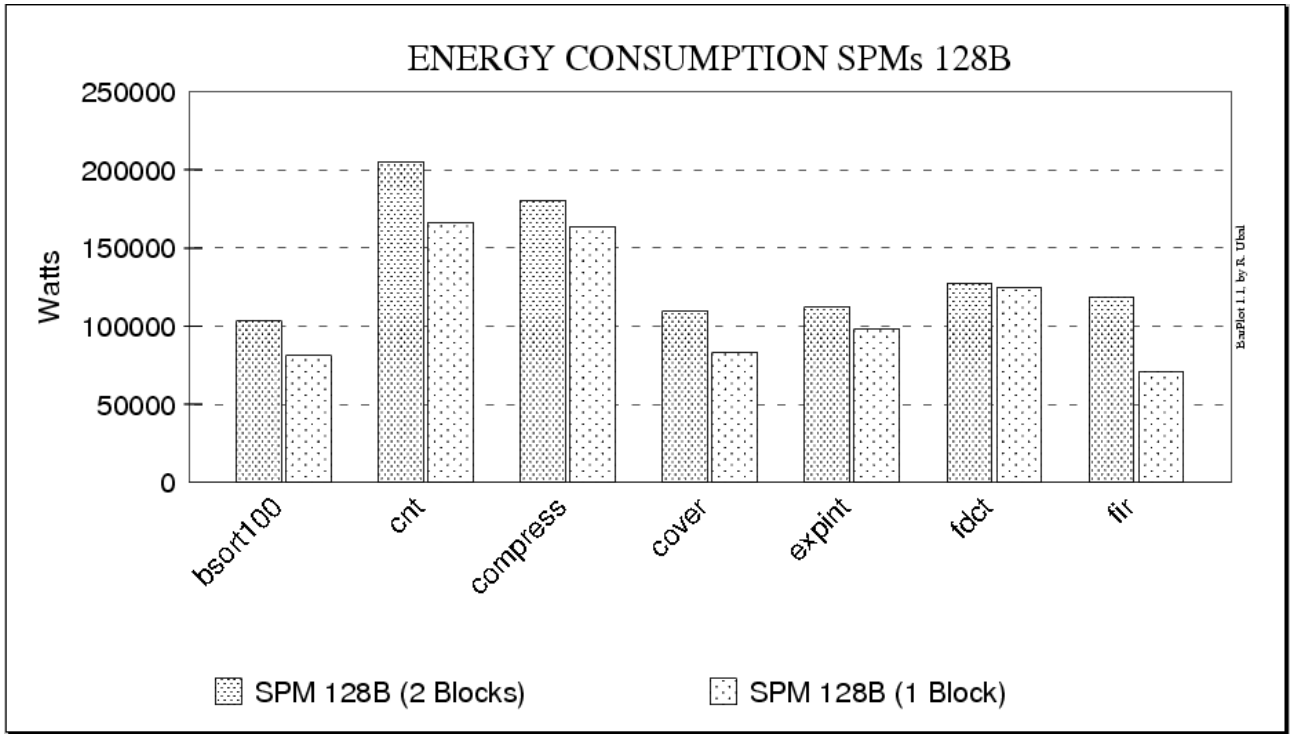

Figura 37 : Comparación de consumo entre SPMs de 128B.

De la misma manera comparando los mismos benchmarks con SPMs de 256B, una con un único bloque de 256B y otra seperada en dos bloques de 128B observamos que la SPM de un único bloque sigue saliendo ganadora en todos los bechmarks con una media de mejora del 20,52% en rendimiento y del 30.12% en consumo enérgetico. Las figuras 38 y 39 muestran estos resultados.

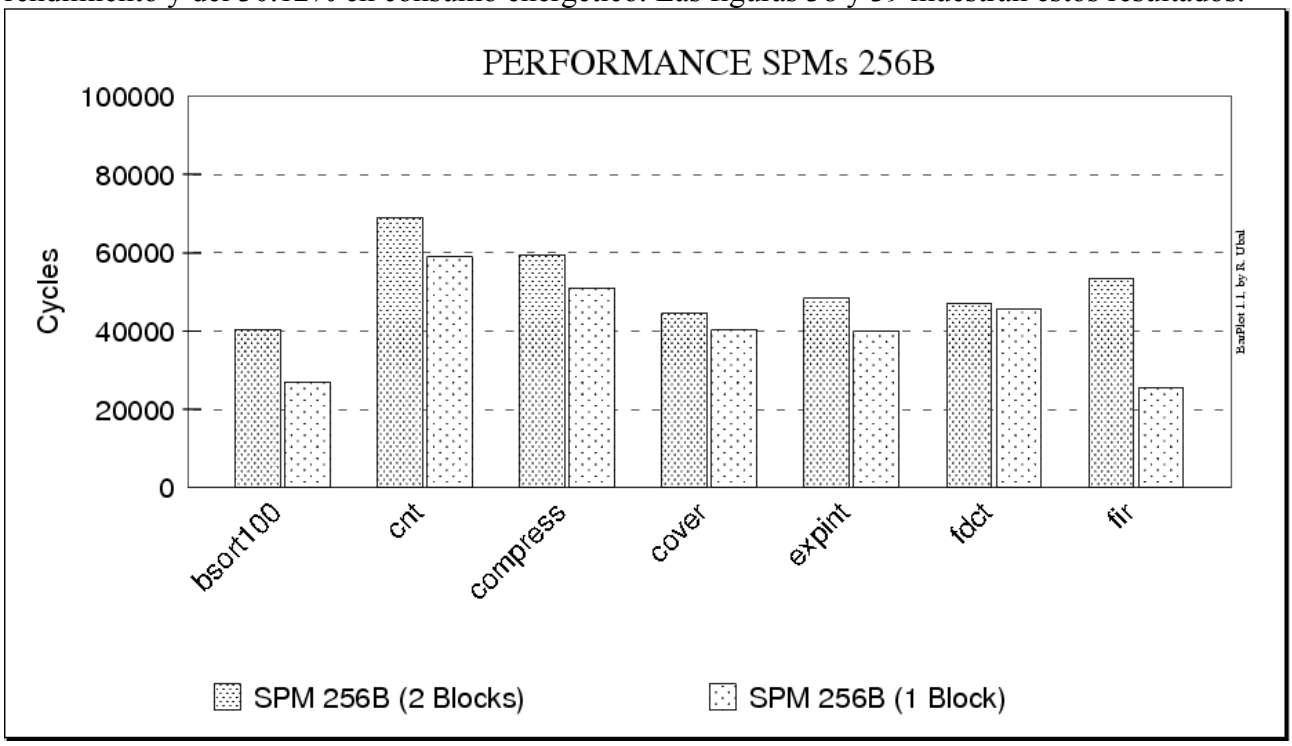

Figura 38 : Comparación de rendimiento entre SPMs de 256B.

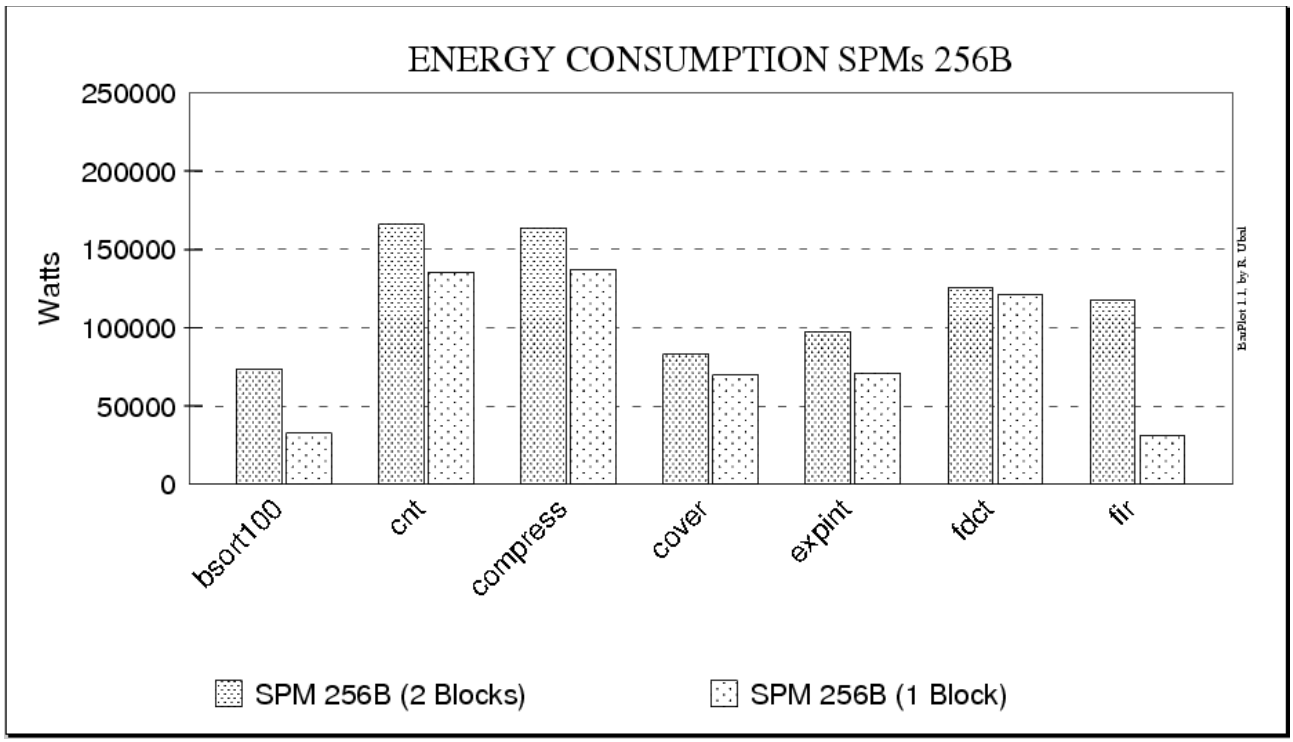

Figura 39 : Comparación de consumo entre SPMs de 256B.

Y por último comparando ambas SPM de 512B, la SPM de un único bloque de 512B vuelve a salir vencedor frente a la SPM de dos bloques de 256B obteniendo unos resultados de mejora de un 17.44% en rendimiento, mostrado en la figura 39 y de un 26,06% en consumo enérgetico, como muestra la figura 40.

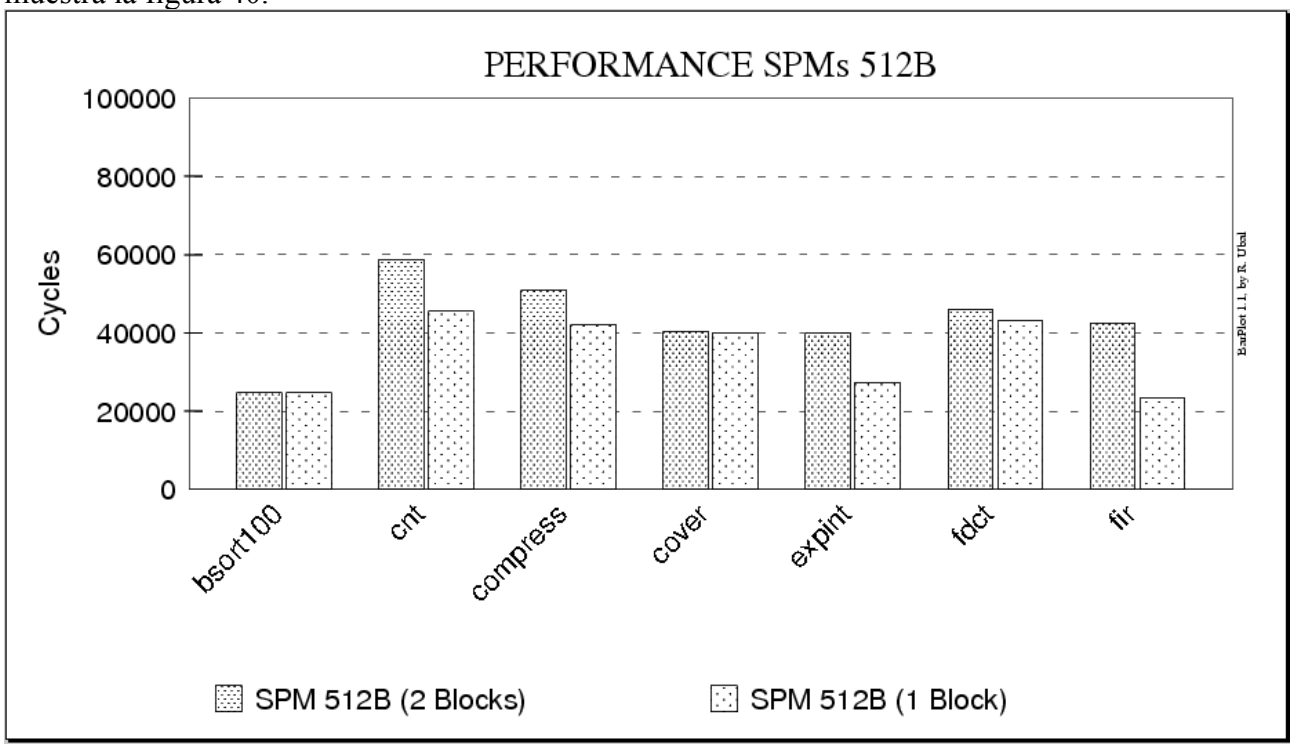

Figura 40 : Comparación de rendimiento entre SPMs de 512B.

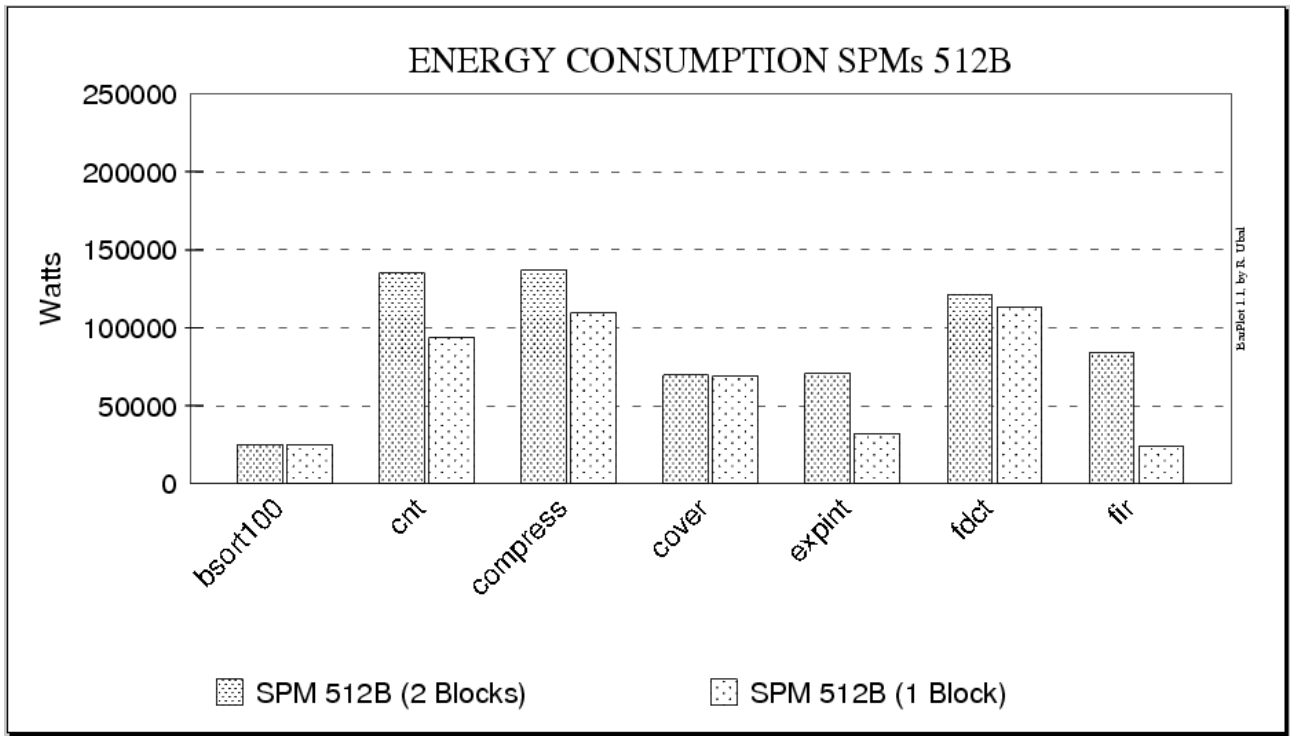

Figura 41 : Comparación de consumo entre SPMs de 512B.

Así pues, para los bechmarks utilizados la utilización de un único bloque obtiene unos resultados mejores, siendo estos un 17,11% en rendimiento y de un 24,88% en consumo enérgetico.

En este proyecto hemos realizado un estudio sobre distintas arquitecturas de memorias on-chip, implementando dos arquitecturas ya existentes en el procesador simplescalar como el bloqueo de cache y la SPM y presentando e implementando una nueva arquitectura basada en una memoria SPM. Esta nueva y original aproximación mejora el control de la scratch-pad memory en procesadores empotrados reduciendo el consumo energético de estos. La idea principal consiste en reducir el sobrecoste resultante de actualizar los contenidos de la SPM mediante una tecnica de carga al vuelo, resultando muy útil al utilizar tecnicas dinamicas de asignación de código en la SPM, donde se requiere cargar dentro de esta una gran cantidad hot spots. Además esta nueva SPM implementada es ortogonal y complementaria a las técnicas de alojamiento de contenidos en la SPM presentadas en disitintos estudios ya comentados en esta memoria, permitiendo obtener un mejor resultado mediante la inclusión de las distintas extensiones propuestas en esta memoria.

BIBLIOGRAFIA

[1] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, M. Balakrishnan, Peter Marwedel: Scratchpad memory: design alternative for cache on-chip memory in embedded systems. CODES 2002: 73-78

[2] Manish Verma, Lars Wehmeyer, Peter Marwedel: Cache-Aware Scratchpad Allocation Algorithm. DATE 2004: 1264-1269

[3] Manish Verma, Klaus Petzold, Lars Wehmeyer, Heiko Falk, Peter Marwedel: Scratchpad Sharing Strategies for Multiprocess Embedded Systems: A First Approach. ESTImedia 2005: 115-120

[4] Nghi Nguyen, Angel Dominguez, Rajeev Barua: Memory allocation for embedded systems with a compile-time-unknown scratch-pad size. CASES 2005: 115-125

[5] Bernhard Egger, Chihun Kim, Choonki Jang, Yoonsung Nam, Jaejin Lee, Sang Lyul Min: A dynamic code placement technique for scratchpad memory using postpass optimization. CASES 2006: 223-233

[6] Bernhard Egger, Jaejin Lee, Heonshik Shin: Scratchpad memory management for portable systems with a memory management unit. EMSOFT 2006: 321-330

[7] Bernhard Egger, Jaejin Lee, Heonshik Shin: Dynamic scratchpad memory management for code in portable systems with an MMU. ACM Trans. Embedded Comput. Syst. 7(2): (2008)

[8] Hyungmin Cho, Bernhard Egger, Jaejin Lee, Heonshik Shin: Dynamic data scratchpad memory management for a memory subsystem with an MMU. LCTES 2007: 195-206

[9] Andhi Janapsatya, Sri Parameswaran, Aleksandar Ignjatovic: Hardware/software managed scratchpad memory for embedded system. ICCAD 2004: 370-377

[10] M. Balakrishnan, Peter Marwedel, Lars Wehmeyer, Nils Grunwald, Rajeshwari Banakar, Stefan Steinke: Reducing Energy Consumption by Dynamic Copying of Instructions onto Onchip Memory. ISSS 2002: 213-218

[11] Francesco Poletti, Paul Marchal, David Atienza, Luca Benini, Francky Catthoor, Jose Manuel Mendias: An integrated hardware/software approach for run-time scratchpad management. DAC 2004: 238-243

[12] Lian Li, Lin Gao, Jingling Xue: Memory Coloring: A Compiler Approach for Scratchpad Memory Management. IEEE PACT 2005: 329-338

[13] Lea Hwang Lee, Bill Moyer, John Arends: Instruction fetch energy reduction using loop caches for embedded applications with small tight loops. ISLPED 1999:267-269

[14] JA Victorio, Enrique F. Torres Moreno, Victor Viñals Yúfera : Vatios: Simulador de Procesador con Estimación de Potencia, XVIII Jornadas de Paralelismo, Zaragoza 2007

[15] Doug Burger and Todd M. Austin: The SimpleScalar Tool Set Version 2.0. Technical Report 1342, Computer Sciences Department, University of Wisconsin--Madison, May 1997.

[16] David Brooks, Vivek Tiwari, Margaret Martonosi: Wattch: a framework for architectural-level power analysis and optimizations. ISCA 2000: 83-94

[17] Tarjan, David; Thoziyoor, Shyamkumar; Jouppi, Norman : CACTI 4.0 , P. HPL-2006- 86 20060606

[18] The Mälardalen WCET research group. The Mälardalen WCET benchmarks homepage , http://www.mrtc.mdh.se/projects/wcet/benchmarks.html

[19] Doosan Cho, Sudeep Pasricha, Ilya Issenin, Nikil D. Dutt, Minwook Ahn, Yunheung Paek: Adaptive Scratch Pad Memory Management for Dynamic Behavior of Multimedia Applications. IEEE Trans. on CAD of Integrated Circuits and Systems (TCAD) 28(4):554-567 (2009)

[20] http://www.mrtc.mdh.se/projects/ wcet/benchmarks.html

[21] Manish Verma, Peter Marwedel: Advanced memory optimization techniques for low-power embedded processors. Springer 2007: I-XII, 1-188

[22] http://quid.hpl.hp.com:9081/cacti/

[23] A. Martí, A. Perles and J. V. Busquets. “Static use of locking caches in multitask preemptive real-time systems”. IEEE Real-Time Embedded System Workshop, December 2001.

[24] X. Vera, B. Lisper and J. Xue. "Data cache locking for higher program predictability". In Proceedings of International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS’03), pages 272–282, June 2003.

[25] X. Vera, B. Lisper and J. Xue. "Data caches in multitasking hard real time systems". In Proceedings of 24th Real-Time Systems Symposium (RTSS’03), December 2003.

[26] http://es.wikipedia.org/wiki/IPhone

[27] http://es.wikipedia.org/wiki/Nintendo_DS

[28] http://www.open-pandora.org/

[29] http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0092b/I14301.html

[30] http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.faqs/ka4271.html

[31] http://infocenter.arm.com/help/index.jsp

[32] http://www.freescale.com

[33] http://www.freescale.com/webapp/sps/site/homepage.jsp?code=PC68KCF

[34] http://infocenter.arm.com/help/index.jsp

[35] John L. Hennessy, David A. Patterson: Computer Architecture - A Quantitative Approach (4. ed.). Morgan Kaufmann 2007

[36] Luis C. Aparicio, Juan Segarra, J.L. Villaroel, Víctor Viñals: Memorias cache en sitemas de tiempo real.

[37] A. Martí, A. Pérez, A. Perles and J. V. Busquets. Using genetics algorithms in content selection for locking-caches. IAESTED International Conference on Applied Informatics, Acta Press, 2001.

[38] A. Martí, J. V. Busquets. Utilización de memorias Cache con bloqueo en sistemas de tiempo real. Tesis Doctoral UPV, 2003.

[39] Isabelle Puaut : Cache analysis vs static cache locking for schedulability analysis in multitasking real-time systems

[40] Isabelle Puaut, Christophe Pais: Scratchpad memories vs locked caches in hard real-time systems: a quantitative comparison. DATE 2007:1484-1489

[41] Isabelle Puaut, David Decotigny: Low-Complexity Algorithms for Static Cache Locking in Multitasking Hard Real-Time Systems. IEEE Real-Time Systems Symposium 2002:114-123