MEJORA DEL RENDIMIENTO DE APLICACIONES MULTIMEDIA EN PROCESADORES ARM

Javier Borrego Pérez
Alvaro Rico Sánchez

Profesores directores:
Marcos Sánchez-Éñez Martín
Daniel Chaver Martínez

Proyecto de Sistema Informáticos 2007/2008
Facultad de Informática
Universidad Complutense de Madrid

Los autores de este proyecto, Javier Borrego Pérez, y Alvaro Rico Sánchez, autorizan a la Univerdidad Complutense, a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la propia memoria, como el código, la documentación y/o el prototipo desarrollado.

Fdo:

Javier Borrego Pérez

Fdo:

Alvaro Rico Sánchez

ÍNDICE

Resumen

Los procesadores de propósito general suponían hasta hace unos años la mayor parte de la fabricación de procesadores. Sin embargo, en las últimas décadas, el aumento de dispositivos electrónicos destinados a una función más concreta (teléfonos móviles, decodificadores de TV y vídeo, GPSs, reproductores MP3 ....), ha hecho disparar la demanda de procesadores dedicados (procesadores empotrados).

Muchas de las técnicas arquitectónicas empleadas en estos procesadores son muy mejorables. El objetivo de este proyecto ha sido investigar y presentar mejoras arquitectónicas para uno de los procesadores empotrados más utilizado comercialmente (ARM[2]). Para ello, nos hemos centrado en la implementación de técnicas de predicción de saltos más agresivas, utilizado el simulador *simplesim-ARM*, y validando dichas técnicas con el conjunto de programas de prueba MiBench.

Palabras clave

Predictor de saltos, sistema empotrado, ARM, MiBench, aplicaciones multimedia

Abstract

General purpose processors supposed the main part of the processor manufacture in the past. However, in the last 2 decades, the growth of electronic devices designed for a more specific function (such as mobile phones, TV/video decoders, GPSs, MP3 players ....) has increased the demand for embedded processors.

Many of the architectural strategies used in current embedded processors can be improved. The aim of this project has been to investigate improvements in the architecture of ARM (one of the main embedded processors nowadays). Specifically, we have been implemented some aggressive branch prediction techniques, using the simplesim-ARM simulator, and we have validated them using a suite of benchmarks specially fit for multimedia applications (MiBench).

Keywords

Branch predictor scheme, embedded system, ARM, MiBench, multimedia applications

1. Introducción

El microprocesador ha sido el motor tecnológico de nuestra era. Este diminuto componente electrónico es el cerebro de la mayor parte de las computadoras que nos ayudan a mejorar nuestra calidad de vida mediante accesorios inteligentes. Como ejemplos de ello: frigoríficos, teléfonos móviles, cámaras digitales, PDAs, aviones, etc. Debido al bajo coste conseguido en la fabricación de microprocesadores, en la actualidad la mayoría de los dispositivos electrónicos presentan un pequeño microprocesador empotrado para realizar una serie de tareas predeterminadas.

Esas “computadoras” que están embebidas o incrustadas en dichos productos son conocidas como **sistemas empotrados**[2] o embebidos (del inglés *embedded*). En general, un SE consiste en un sistema con microprocesador cuyo hardware y software están específicamente diseñados y optimizados para resolver un problema concreto eficientemente. Debido al creciente mercado de dispositivos que utilizan estos sistemas, se ha hecho necesaria una revisión y mejora en las técnicas arquitectónicas empleadas, que son manifiestamente mejorables.

Los procesadores empotrados actuales son capaces de leer y ejecutar más de una instrucción por ciclo, utilizando técnicas de *pipelining*[15] para explotar el paralelismo en la ejecución de instrucciones, mejorando con ello el rendimiento. Las instrucciones de salto suponen un inconveniente para la obtención de un rendimiento alto. Estas instrucciones provocan la parada del flujo de ejecución del programa hasta que no se conoce la dirección destino y no se sabe si el salto es tomado (si salta a la dirección destino) o no. Como el procesador ha estado varios ciclos sin leer ninguna instrucción, nos encontramos con que varias etapas del pipeline están vacías y que la ventana de instrucciones (donde están las instrucciones pendientes de ser ejecutadas) se ve reducida. Como resultado de produce una degradación importante del rendimiento del procesador, ya que éste tendrá menos instrucciones donde elegir para lanzar a ejecutar.

Para evitar la penalización introducida por los cambios en el flujo de ejecución de un programa, una solución adoptada por muchos procesadores consiste en tener técnicas precisas de predicción de los saltos condicionales. La utilización de dichas técnicas pretende reducir la penalización producida por los saltos, haciendo *prefetching* y ejecutando instrucciones del camino destino antes que el salto sea resuelto. Esta circunstancia es conocida como *ejecución especulativa*, ya que se ejecutan instrucciones sin saber si son las correctas en el orden del programa. Para ello se intenta predecir si un salto será efectivo (si será tomado) o no, así como realizar el cálculo de la dirección destino antes que la instrucción de salto llegue a la unidad de saltos del procesador. De esta manera se intenta no romper el flujo de ejecución del programa, leyendo continuamente instrucciones de la cache.

El objetivo de este proyecto ha sido investigar y presentar mejoras arquitectónicas para uno de los procesadores empotrados más utilizado comercialmente (ARM), implementarlas en el simulador de este procesador (*simplescalar-ARM*), y validarlas con un conjunto de programas de prueba de aplicaciones multimedia. Concretamente las mejoras propuestas estarán

enfocadas hacia ***nuevas técnicas de ejecución especulativa de instrucciones*** (predicción de saltos).

Nuestro trabajo ha consistido, por tanto, en la implementación de una serie de técnicas de predicción dinámica de saltos, algunos más simples como el Agree, otros más complejos y agresivos como el Yags, Skewed y Bi-Mode, e incluso un esquema más innovador, como el que presenta el predictor Perceptrón.

Tras la aplicación de estas nuevas técnicas, hemos observado y contrastado una mejora del comportamiento del procesador, tanto a efectos de tasa de aciertos de salto como a efectos del CPI, respecto a los resultados obtenidos con los métodos de predicción más simples de que disponía el Simplescalar ARM (predictor Bimodal, gshare y esquemas tipo McFarling), que analizaremos en los siguientes puntos.

2. Sistemas Empotrados

Un Sistema Empotrado[14] (en adelante, lo denominaremos SE), es un sistema computador de propósito específico construido en un dispositivo mayor, que con sus elementos externos desarrolla una función determinada de manera autónoma.

Normalmente un SE interactúa continuamente con el entorno para vigilar o controlar algún proceso mediante una serie de sensores. Su hardware se diseña normalmente a nivel de chips, o de interconexión de PCBs (placas de circuito impreso), buscando la mínima circuitería y el menor tamaño para una aplicación particular. En general, un SE simple contará con un microprocesador, memoria, unos pocos periféricos de E/S y un programa dedicado a una aplicación concreta almacenado permanentemente en la memoria. El término embebido o empotrado hace referencia al hecho de que el microcomputador está encerrado o instalado dentro de un sistema mayor y su existencia como microcomputador puede no ser aparente. Un usuario no técnico de un sistema empotrado puede no ser consciente de que está usando un sistema computador. En algunos hogares las personas, que no tienen por qué ser usuarias de un ordenador personal estándar (PC), utilizan del orden de diez o más dispositivos empotrados cada día.

Por tanto, un SE tiene como integrante fundamental al computador, ya sea en sus versiones de microprocesador o de microcontrolador. Lo esencialmente importante es que su funcionamiento se lleve a cabo mediante programación. Pero como el componente computador no se puede ver como tal, se debe incluir los datos de entrada, y obtener los datos de salida, de un modo no convencional. Esta situación implica que al sistema computador hay que añadirle componentes hardware para tal efecto.

En la figura 3.1 podemos apreciar la arquitectura típica de un SE. Podemos ver la CPU y la memoria, junto con una amplia variedad de interfaces que permite al sistema medir, manipular e interactuar con el entorno exterior. En esta figura podemos ver que existen dos tipos de comunicación: una comunicación en el interior del sistema (que se realizará a través de un bus de sistema), y una comunicación externa (del sistema con el exterior, a través de un conjunto de sensores).

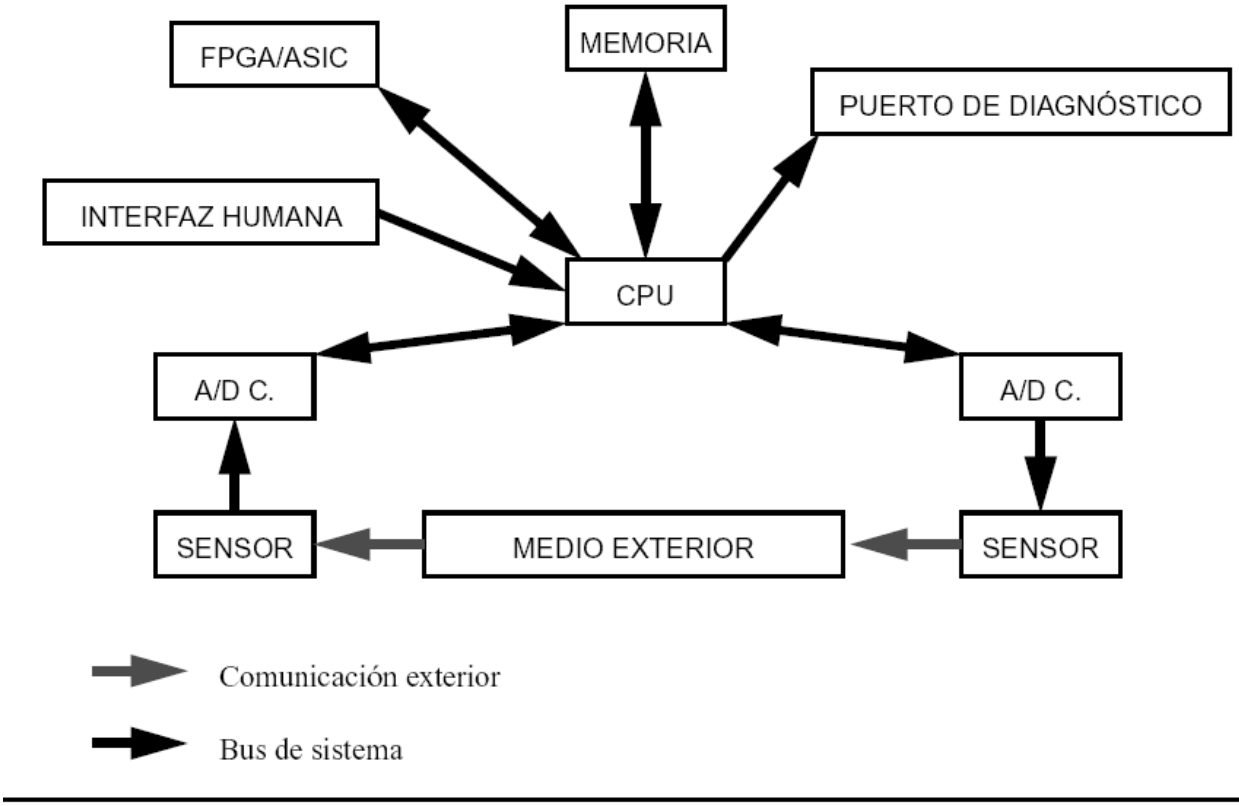


Fig. 2.1 Arquitectura típica de un Sistema Empotrado

Las principales características de un SE son las siguientes:

- Emplean una combinación de recursos hardware y software limitada, para realizar una función específica.
- Realizan una única función, o un conjunto muy limitado de funciones (al contrario que los sistemas de propósito general).
- La potencia, el coste y la realizabilidad (poder ser fabricados por millares o millones de unidades) son los principales factores de diseño.

Además de las anteriores, inherentes a cualquier SE, para que un sistema de este tipo sea útil, deberá tener las siguientes características:

- **Concurrencia**. Los componentes del sistema funcionan simultáneamente, por lo que el sistema deberá operar a la vez.
- **Fiabilidad y seguridad**. El sistema debe ser fiable y seguro frente a errores, ya que puede requerir un comportamiento autónomo. El manejo de estos errores puede ser vía hardware o software, aunque la utilización software nos dará un sistema menos robusto.
- **Interacción con dispositivos físicos**. Los sistemas empotrados interaccionan con el entorno a través de dispositivos E/S no usuales, por lo que suele ser necesario un acondicionamiento de las diferentes señales.
- **Robustez**. El sistema empotrado se le impondrá la necesidad de la máxima robustez ya que las condiciones de uso no tienen porqué ser

"buenas", sino que pueden estar en el interior de un vehículo con diferentes condiciones de operación.

- **Bajo consumo**. El hecho de poder utilizar el sistema en ambientes hostiles puede implicar la necesidad de operaciones sin cables. Por lo tanto, un menor consumo implica una mayor autonomía de operación.
- **Precio reducido**. Esta característica es muy útil cuando estamos hablando de características de mercado. Esta situación no es nada inusual en el campo de los sistemas empotrados ya que tienen una gran cantidad de aplicaciones comerciales, tanto industriales como de consumo.
- **Pequeñas dimensiones**. Las dimensiones de un sistema empotrado no dependen sólo de sí mismo sino también del espacio disponible en el cual dicho sistema va a ser ubicado.

Entre las aplicaciones y finalidades de los sistemas empotrados, podemos encontrar diferentes clasificaciones, una de las principales atendiendo a su interacción con el entorno:

- Sistemas reactivos.- son aquellos sistemas que siempre interactúan con el exterior, de tal forma que la velocidad de operación del sistema deberá ser la velocidad del entorno exterior (controladores de airbag, sistemas sensores, etc).
- Sistemas interactivos.- son aquellos sistemas que siempre interactúan con el exterior, de tal forma que la velocidad de operación del sistema deberá ser la velocidad del propio sistema empotrado (PDAs, smartphones, videojuegos portátiles, etc).
- Sistemas transformacionales.- son aquellos sistemas que no interactúan con el exterior, únicamente toma un bloque de datos de entrada y lo transforma en un bloque de datos de salida, que no es necesario en el entorno (dispositivos de red como routers, switches..).

2.1 Arquitectura ARM

La familia de procesadores ARM[2] (*Advanced RISC Machines*) de propósito general, es uno de los núcleos más conocidos en el mundo, por su utilización en gran cantidad de dispositivos portátiles de consumo.

La arquitectura ARM se basa en los principios del repertorio de instrucciones RISC (*Reduced Instruction Set Architecture*), su repertorio de instrucciones y su mecanismo de decodificación son sumamente sencillos, comparados con computadores basados en CISC (*Complex Instruction Set Architecture*). Esta simplicidad se traduce en un alto IPC y una enorme capacidad de respuesta en tiempo real desde un chip reducido y de bajo coste.

Dentro de la familia ARM existen varias versiones: ARM6, ARM7, ARM9, ARM10, y ARM11. Además, existen algunas extensiones a estos núcleos, como THUMB, JAZELLE, ElSegundo , etc, cada una de ellas implementa alguna modificación sobre la arquitectura original. De todas ellas, la versión más extensamente utilizada es la versión *ARM7TDMI*.

El procesador ARM7TDMI[2] emplea una única estrategia arquitectural llamada THUMB[2], que lo hace muy apropiado para aplicaciones voluminosas en un entorno con restricciones en el empleo de recursos hardware. El interface de memoria esta diseñado para obtener un rendimiento alto sin incurrir en costes altos en el sistema de memoria. Vemos el diagrama del núcleo del procesador en la figura 3.2.

THUMB reduce la longitud de instrucción a 16 bits, reduciendo el gasto de memoria notablemente. Esencialmente, el procesador ARM7TDMI utiliza dos repertorios: el repertorio estándar del ARM y un repertorio del 16 bits THUMB, exisitiendo gran facilidad para el cambio entre ambos modos. Esto se hace posible gracias a que la extensión THUMB opera sobre los mismos registros de 32 bits de la arquitectura ARM, por tanto no es necesario ninguna modificación en la arquitectura. Sí se añade, en cambio, un pequeño módulo en la etapa de decodificación de la instrucción del pipeline, donde se identifica si se trata de una instrucción de código nativo ARM, o una de la extensión THUMB.

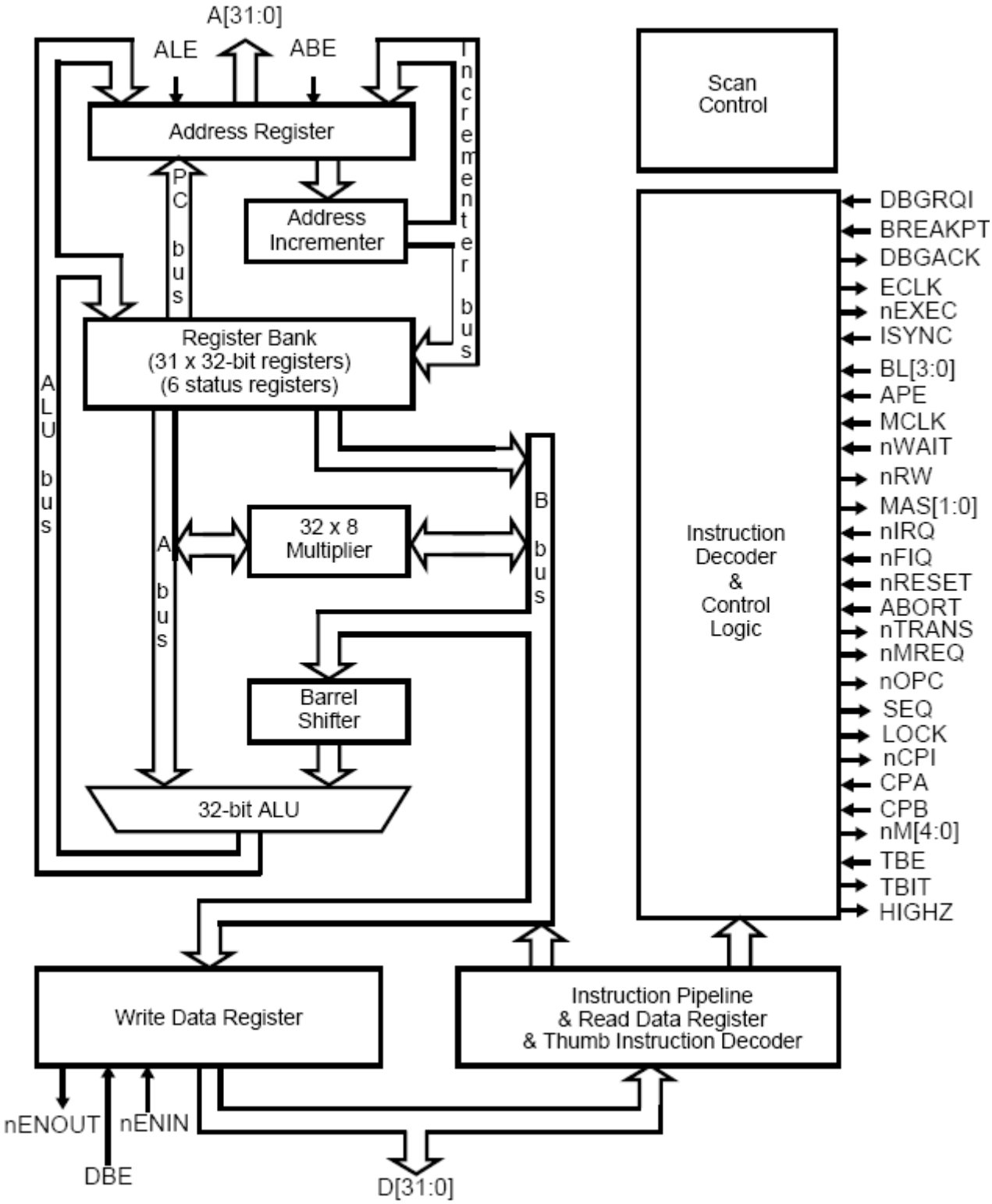


Fig. 2.2 Diagrama del núcleo del procesador ARM7TDMI

El formato de memoria utilizado por el ARM7TDMI consiste en una secuencia de bytes ascendente, comenzando por el 0. El direccionamiento en memoria se gestionará por múltiplos de 4 bytes (palabras de memoria), siendo la primera palabra la que comprende los bytes del 0 al 3, y así sucesivamente. Dichas palabras se pueden gestionar en al ser almacenadas en memoria en formato *big-endian* (del bit 0 al 31) o en formato *little-endian* (del bit 31 al 0), como vemos en las figuras 3.3 y 3.4.

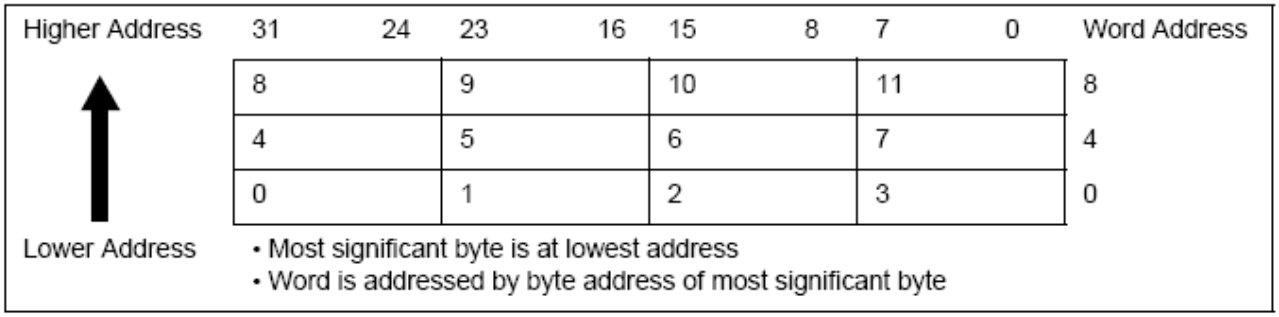


Fig. 2.3 Direccionamiento *big-endian* en palabras de memoria

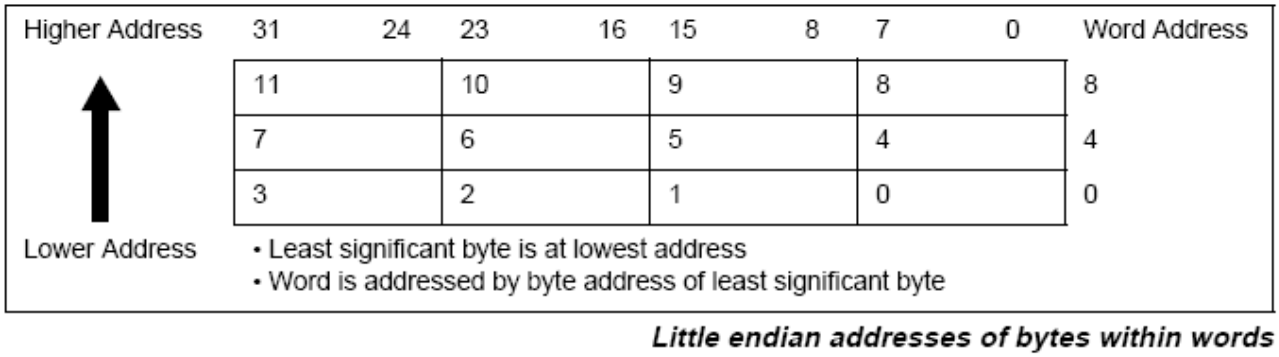


Fig. 2.4 Direccionamiento *little-endian* en palabras de memoria

Como hemos dicho antes, las instrucciones pueden ser o bien de 32 bits de longitud (para el estado ARM), o de 16 bits (para el estado THUMB). Están soportados los tipos de datos de 8 bits (1 byte), 16 bits (media palabra), y 32 bits (palabra). Las palabras deben estar alineadas en direcciones de memoria múltiplos de 4, y las medias palabras en direcciones de memoria múltiplos de 2.

El ARM7TDMI dispone de 37 registros (31 registros de 32 bits de propósito general, y 6 registros de estado), pero no todos ellos están disponibles al mismo tiempo. El estado del procesador y el modo de operación dictan cuáles de ellos están disponibles en cada momento.

El procesador soporta 7 modos de operación distintos: *User* (usr), el modo normal de operación; *FIQ* (fiq), diseñado para soportar transferencias de datos; *IRQ* (irq), para manejo de interrupciones; *Supervisor* (svc), modo protegido para el Sistema Operativo; *Abort mode* (abt); *System* (sys), modo de usuario privilegiado para el SO; *Undefined* (und), cuando se introduce una instrucción desconocida. Los cambios de modo pueden ocurrir bajo control de programa, bajo interrupciones externas, o bajo procesamiento de excepciones.

2.2 Simulador *simplesim-arm*[1]

Expondremos ahora brevemente el funcionamiento del simulador *simplesim*-arm[1], herramienta que hemos utilizado para implementar y validar las mejoras arquitectónicas introducidas. Se trata de la versión v4.0 del simulador SimpleScalar, que tiene las siguientes características, en cuanto a la emulación de instrucciones para procesadores ARM se refiere:

- Soporte para repertorio de instrucciones enteras en la versión ARM 7.
- Soporte para repertorio de instrucciones en punto flotante (*flotant point accelerator. FPA*).
- Soporte para llamadas LINUX/ARM al sistema, implementadas por el simulador.
- Soporte para repertorio de instrucciones de microcódigo *CISC*.
- Implementación del mecanismo de ejecución de instrucciones siguiendo el modelo de segmentación de Intel SA-1 (ifgura 5.5), con 5 etapas en el pipeline, y una jerarquía de memoria de 2 niveles (soporte para cache, en el módulo cache.h/.c.

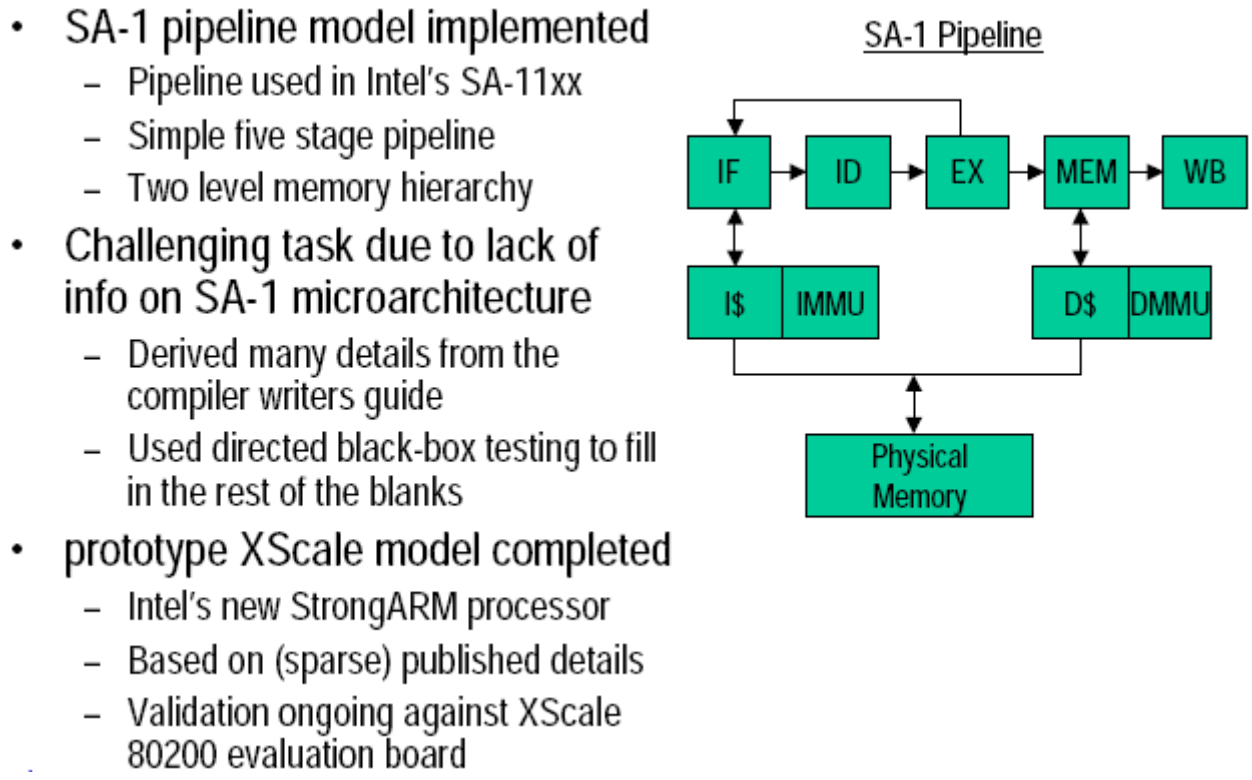


figura 2.5 Modelo de ejecución de instrucciones implementado

Esta distribución del simulador incluye varias modalidades de simulación, las cuales se pueden organizar en orden creciente de detalle y complejidad, y a

su vez en orden decreciente de velocidad de simulación, como vemos en la figura 5.5. Cada módulo sim-* representa el núcleo completo de un simulador.

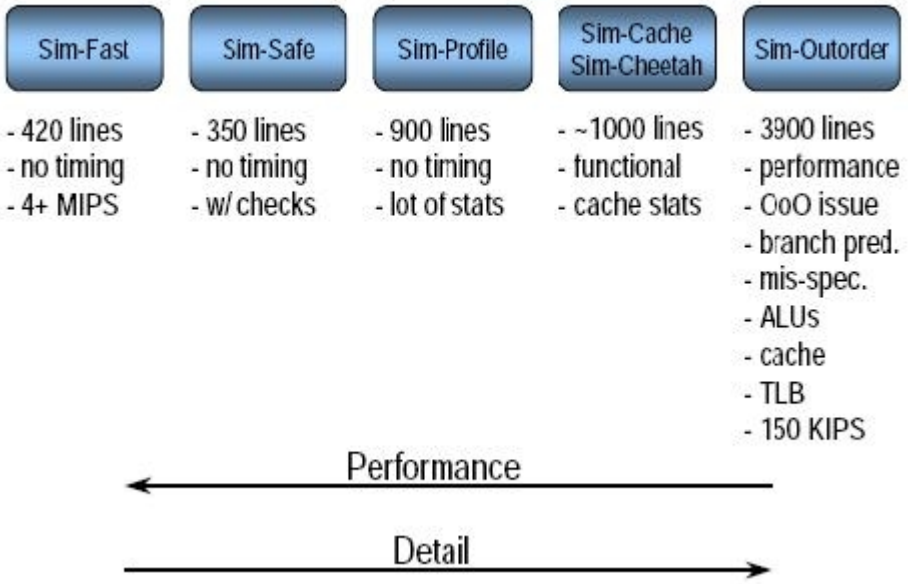


Fig. 2.6 Módulos de simulación de la distribución *simplesim-arm*

Además de los módulos anteriores, la distribución de simplesim-arm incluye también un pequeño módulo para simulación de un predictor de saltos (ficheros fuente *sim-bpred.c*, *bpred.c* y *bpred.h*), y es éste módulo el que hemos modificado implementando las técnicas de predicción mencionadas al principio del documento.

Para obtener una visión más general podemos dividir la estructura del simulador en una serie de capas bien definidas.

1. En la capa más alta encontramos los programas de los usuarios, por ejemplo, como más adelante veremos, los benchmarks que vamos a utilizar para obtener el rendimiento de los predictores entrarían dentro de esta capa.

2. Un nivel más por debajo encontramos la interface programa / simulador. Esta capa sirve de intermediaria entre la capa de usuario y la capa siguiente, la del simulador. Esta capa está formada por el módulo de llamadas POSIX al sistema y por un bus denominado SimpleScalar ISA.

3. La tercera capa es el núcleo funcional formado por la definición de la máquina y un proxy manejador de llamadas al sistema (procedentes de la capa superior).

4. Por último nos encontramos la capa donde se encuentra el verdadero núcleo del simulador donde encontramos ,entre otros módulos, la memoria, los registros y la caché.

Toda esta estructura se puede ver resumida en la siguiente figura:

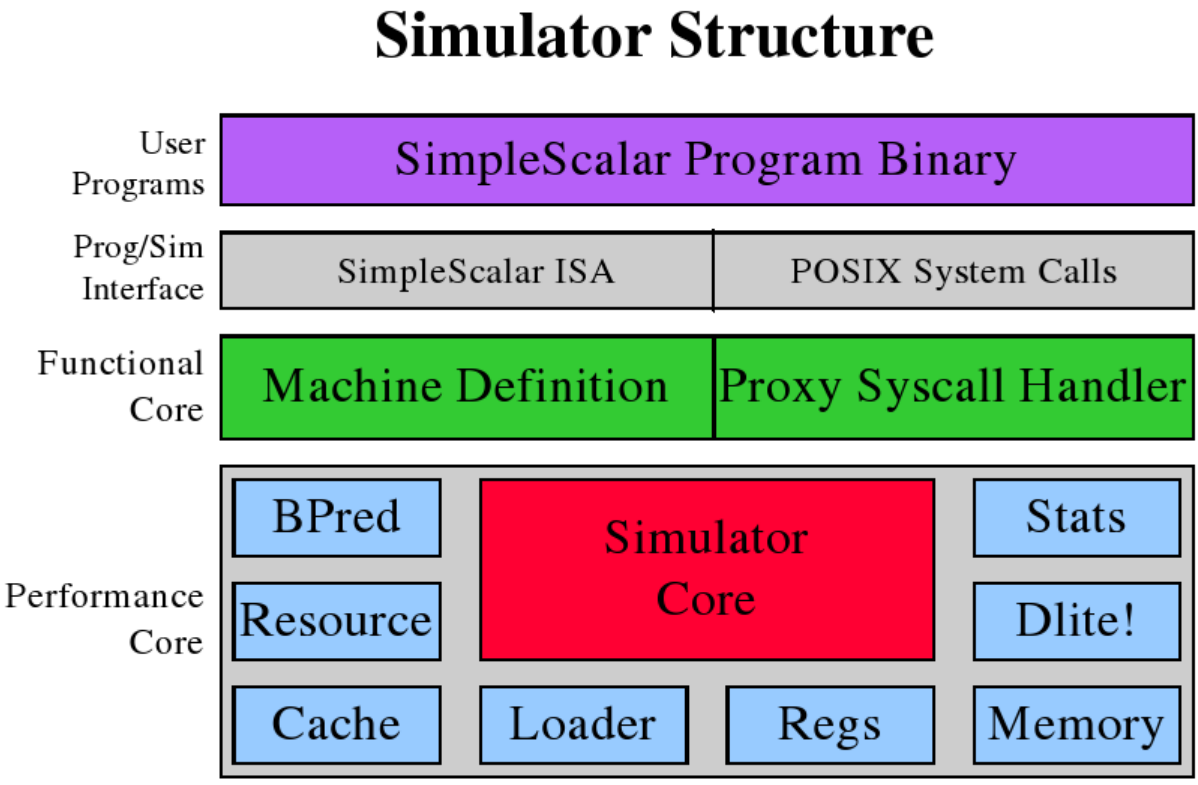


Fig 2.7

Para ejecutar los tests y poder obtener las medidas de mejora de rendimiento del procesador y también los resultados de mejora de la tasa de aciertos en las instrucciones de salto condicional, hemos utilizado el módulo *sim-outorder*, puesto que es el módulo más completo, dispone de ejecución especulativa y ejecución de instrucciones fuera de orden entre otras características, dos aspectos muy interesantes y necesarios para las mediciones que hemos realizado. No obstante, para ir realizando pequeñas pruebas durante la implementación de los predictores, hemos encontrado muy útil el módulo sim-bpred (que nos da la tasa de acierto del predictor utilizado).

Por último, en la figura 5.8 podemos observar claramente cómo funciona el mecanismo de lanzamiento de ejecución de instrucciones, así como la jerarquía de memoria del módulo sim-outorder.

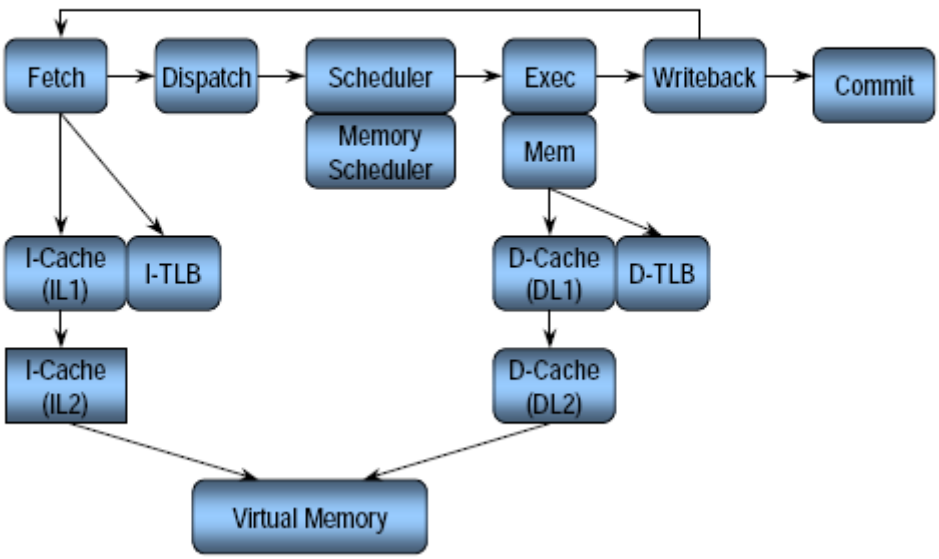

Out-of-Order Issue Simulator
Fetch
Dispatch
Scheduler
Exec
Writeback
Commit
Memory Scheduler
Mem
I-Cache (IL1)
I-TLB
D-Cache (DL1)
D-TLB
I-Cache (IL2)
D-Cache (DL2)
Virtual Memory

Fig 2.8

3. Predictores de Salto

Las técnicas de predicción de salto[16] se pueden dividir, básicamente, en dos tipos:

- Las que realizan predicción estática[16].
- Las que realizan predicción dinámica[16].

La diferencia radica en el momento en el que se realiza la predicción, en el caso de la predicción estática se realiza en tiempo de compilación, mientras que la dinámica se realiza en tiempo de ejecución. Ambas tienen sus ventajas y sus inconvenientes. Por ejemplo con la predicción dinámica se consigue mayor precisión a cambio de un coste en hardware mayor, mientras que la estática tiene menos precisión, no requiere tanto hardware, pero a cambio en algunas implementaciones puede llegar a incrementar la longitud del ejecutable considerablemente (hasta un 30%).

3.1 Técnicas de predicción estática

Existen diversas técnicas de predicción estática de saltos. Las más sencillas se basan en las propiedades estáticas de los saltos, como su código de operación o su dirección destino, mientras que las más complicadas se basan en un proceso de *profiling*, es decir, en realizar ejecuciones previas para obtener medidas que permitan deducir estáticamente el comportamiento del salto.

Las técnicas más relevantes de predicción estática son:

- Predecir todos los saltos como tomados: esta técnica es la más sencilla, pero obviamente su precisión es bastante pobre.
- Predicciones basadas en el código de operación: está basada en estudios que dicen que según el tipo de salto que se realiza, la posibilidad que sea tomado o no es diferente.
- Predecir los saltos en función de su dirección: por ejemplo, los saltos “hacia atrás” predecirlos como tomados y los saltos “hacia adelante” predecirlos como no tomados. Esta técnica está basada en el hecho que una gran mayoría de los saltos “hacia atrás” corresponden a bucles, y por lo tanto serán tomados todas las veces que el bucle se ejecute menos una, en cambio los saltos hacia adelante corresponden más a estructuras if-then-else.

Esta técnica funcionará bien en programas con muchos bucles, mientras que no tendrá gran eficacia en programas con un comportamiento irregular de los saltos.

3.2 Técnicas de predicción dinámica

En estas técnicas, la predicción sobre el resultado de un salto se basa en información conocida sólo en tiempo de ejecución, (al contrario de la predicción estática).

Dos estructuras son necesarias para realizar una predicción dinámica:

- Branch History Table (BHT): Es una tabla donde se guarda información sobre las últimas ejecuciones de los saltos, esta información se refiere a si el salto ha sido efectivo o no. A partir de esta información se predice si el siguiente salto será tomado o no.
- Branch Target Address Cache (BTAC): Es una tabla donde se almacena la dirección destino de los últimos saltos ejecutados. De esta manera, cuando un salto es predicho como tomado, se mira si está en la tabla, y si es así, se obtiene la dirección destino de ella, así se puede calcular rápidamente la dirección destino, incluso en saltos indirectos.

La base de la mayoría de los predictores dinámicos es el llamado Branch Target Buffer. Esta estructura combina las dos mencionadas anteriormente (BTAC y BHT). Cada entrada contiene los bits necesarios para realizar la predicción de efectividad, así como la posible dirección destino.

A partir de esta estructura básica se han ido diseñando los distintos predictores, desde el más sencillo que para cada salto guarda unos bits de información sobre su historia más reciente, hasta los más complicados que usan dos niveles de predicción. Últimamente se están proponiendo también unos predictores híbridos que combinan distintas técnicas (estáticas y dinámicas), escogiendo en cada momento la salida del que se cree que es el mejor (el que lleva más aciertos hasta entonces).

Un aspecto a considerar es la influencia de los fallos de predicción en el rendimiento de los procesadores. Debido a ello nos pueden aparecer dos tipos de penalizaciones:

• Misfetch penalty: Los procesadores, para mantener el rendimiento en cada ciclo hacen el fetch de las instrucciones de la Icache. Si aparece una instrucción de salto, y la predicción de salto tomado se hace en la etapa de decodificación, las instrucciones leídas de la Icache en ese ciclo y en el anterior no son válidas y han de ser descartadas. Esto provoca que se introduzca una burbuja en el procesador.

• Mispredict penalty: Al realizar la predicción de una instrucción de salto, el procesador lee en el siguiente ciclo de la Icache las instrucciones correspondientes al bloque destino del salto. Cuando la instrucción de salto es ejecutada en su unidad funcional se comprueba si la predicción realizada anteriormente ha sido correcta. Si ha habido un fallo en la predicción, el procesador deberá realizar un vaciado del pipeline, quitando todas las instrucciones del camino incorrecto pendientes de ser ejecutadas y restaurando el estado de los registros como estaba antes de predecir el salto. A su vez deberá realizar el fetch de las instrucciones del camino correcto. Esto supone una degradación importante en el rendimiento del procesador, ya que al vaciarse el pipeline se introduce una nueva burbuja.

Para comparar distintas arquitecturas de predicción de salto necesitamos unas medidas que nos indiquen el rendimiento. Una, la más sencilla, podría ser la precisión de aciertos, es decir, el porcentaje de saltos que han sido bien predichos. Otras, más complicada, se basa en el misfetch penalty y en el misprediction penalty.

Otro punto a tener en cuenta en las estructuras de predicción dinámica que guardan información referente a la ejecución de los saltos es el comportamiento de los mismos y su frecuencia de ejecución. Esto es debido a que no es lo mismo tener pocos saltos que se ejecutan muchas veces (como en programas que tienen muchos bucles) que muchos saltos que se ejecutan pocas veces y que por lo tanto será más difícil analizar su comportamiento.

Como se ha comentado antes, un problema importante de los predictores de dos niveles es el hecho que no siempre podremos tener un registro de historia por salto, pues esto sería muy costoso en términos del hardware a utilizar. En estos casos se usan los j bits menos significativos de la dirección de la instrucción de salto para escoger el registro de historia que corresponde al salto. Eso significa que los saltos que tengan los j bits menos significativos iguales compartirán el registro de historia, e indexarán la misma entrada en la tabla de patrones. Este hecho es conocido como *aliasing*.

Se han realizado numerosos estudios sobre este problema y dividen el *aliasing* en tres tipos:

- *constructivo*, si el aliasing mejora la predicción respecto a un método en el que hay un registro de historia por dirección,
- *destructivo*, si el aliasing reduce la precisión en las predicciones e
- *inofensivo* si el *aliasing* no cambia la precisión.

El aliasing que normalmente se encuentra en las aplicaciones es el destructivo, y por lo tanto el rendimiento de la predicción se degrada considerablemente. Para evitarlo, hay que gastar más hardware en registros de historia y tener uno por cada salto. Pero, en un dispositivo portátil, de reducidas dimensiones, y que cuenta con escasos recursos hardware, esta opción de aumentar los recursos hardware no es viable en absoluto. en lugar de ello, propondremos, implementaremos, y validaremos un conjunto de técnicas depuradas y encaminadas a minimizar, sino eliminar, este tipo de aliasing destructivo que provoca una degradación importante en el rendimiento del procesador.

A continuación describimos los predictores que hemos usado para medir el rendimiento, los dos primero vienen en la implementación del Simulador y el resto los hemos desarrollado.

Predictor Bimodal [7]

El más simple de todos los que utilizan predicción dinámica. Se trata de una tabla de contadores saturados, también conocida como PHT (Pattern history table) de dos bits los cuales pueden representar 4 estados: fuertemente no tomado, débilmente no tomado, débilmente tomado, fuertemente tomado. Cada vez que se produce un salto se acude a esta tabla indexada por la dirección del salto. Si el contador correspondiente a la entrada seleccionada indica “tomado”

(estados 3 y 4) el salto se predice como tal, en caso contrario el salto será “no tomado”.

Uno de los beneficios principales de este predictor de dos bit es en los bucles, cuanto más grandes mejor, ya que sólo se producen un fallo en el último salto, el resto serán saltos “tomados”. Este predictor obtiene buenos resultados para aplicaciones con saltos fuertemente sesgados; por el contrario gran problema que presenta es lo que se conoce como “aliasing” el cual trataremos de reducir esquema tras esquema que vayamos presentando.

Como veremos más adelante en la sección de resultados, este esquema de predicción demuestra sus limitaciones, no obstante bastante satisfactorios si tenemos en cuenta su simplicidad.

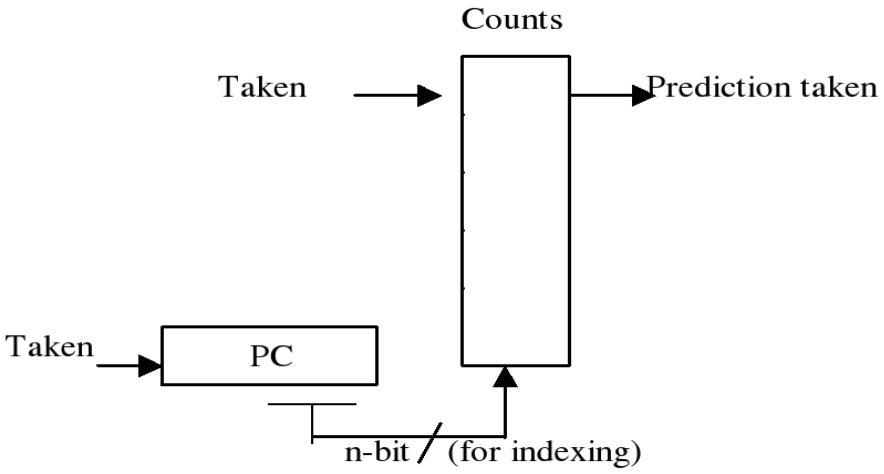


Fig 3.1

Predictor Gshare [12][13]

Se trata del primer esquema de dos niveles que trata de solucionar el problema del aliasing. En este caso al hablar del predictor Gshare tenemos ya definidos los parámetros que describen las estructuras de datos, siendo un esquema de historia global con un ancho de registro W y una tabla de contadores de 2^W entradas. En los esquemas de dos niveles que usan la historia de los saltos lo habitual es acceder a la tabla de contadores utilizando n bits del registro de historia concatenados a m bits de la dirección de la instrucción de salto siendo m+n = W. En nuestro caso tenemos una peculiaridad: se realiza una función O-exclusiva entre los bits del registro de historia y de la dirección de salto. El cambio de la concatenación por la XOR ha demostrado en diversos estudios un mejor comportamiento al minimizar los efectos del aliasing.

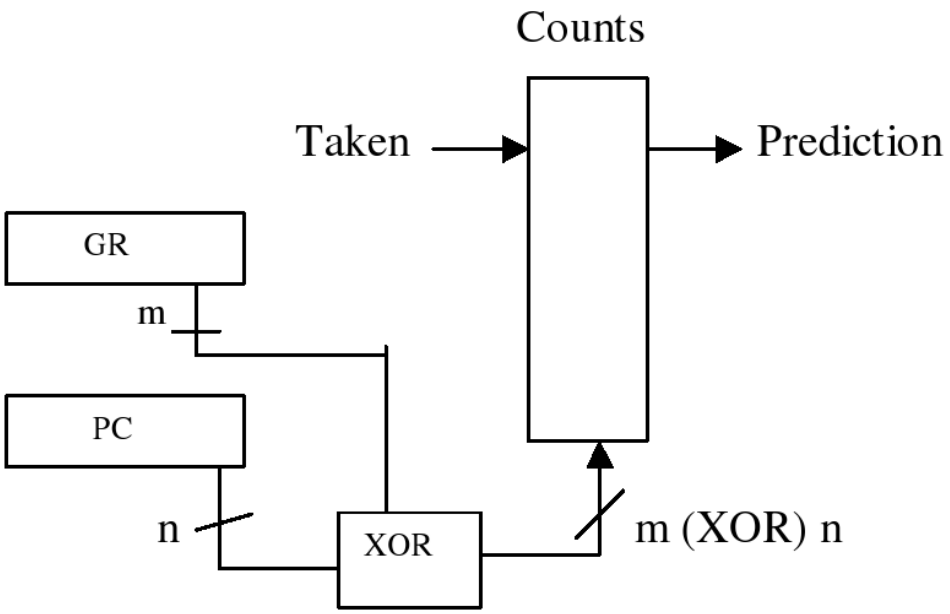


Fig 3.2

Uniendo la ventaja que obtiene al hacer uso de la historia reciente de los saltos junto con el modo de acceso (XOR) este predictor presenta un alto rendimiento para la simplicidad de su estructura.

Predictor Agree [9]

La idea tras este predictor es que en muchos saltos su comportamiento queda muy marcado por el que demostró la primera vez que se introdujo en la BTB, es decir, mantiene ese comportamiento.

Este predictor asigna un bit de sesgo a cada entrada de la BTB de acuerdo con el resultado del salto justo antes de ser introducido en dicha tabla. La PHT ahora cambia su información de “tomado” y “no tomado” a “de acuerdo” y “no de acuerdo”. La idea tras este predictor es que en muchos saltos su comportamiento queda muy marcado por el que demostró la primera vez que se introdujo en la BTB, es decir, mantiene ese comportamiento. Debido a esto la mayoría de las entradas de la PHT mostrarán el valor “de acuerdo” (valor 3 del contador saturado) y por lo tanto en caso de ocurrir que a otro salto le correspondiera puntualmente la misma entrada (aliasing), se trataría más bien de “aliasing neutro” pues no conllevaría a una predicción errónea.

Este predictor es uno de los primeros esquemas que sustituyen el aliasing “destructivo” por un aliasing “neutro”.

Sin embargo este esquema tiene sus incovenientes: nada nos asegura que la primera vez que entra un salto en la BTB su comportamiento vaya a corresponder con su sesgo o comportamiento que mostró la primera vez que se introdujo en la BTB. Cuando esto ocurre, el bit de sesgo va permanecer hasta que el salto abandone la BTB de forma que va a contaminar la PHT con información “no de acuerdo” dando lugar a errores en la predicción. Además si un salto no se encuentra en la BTB no hay disponible predicción alguna.

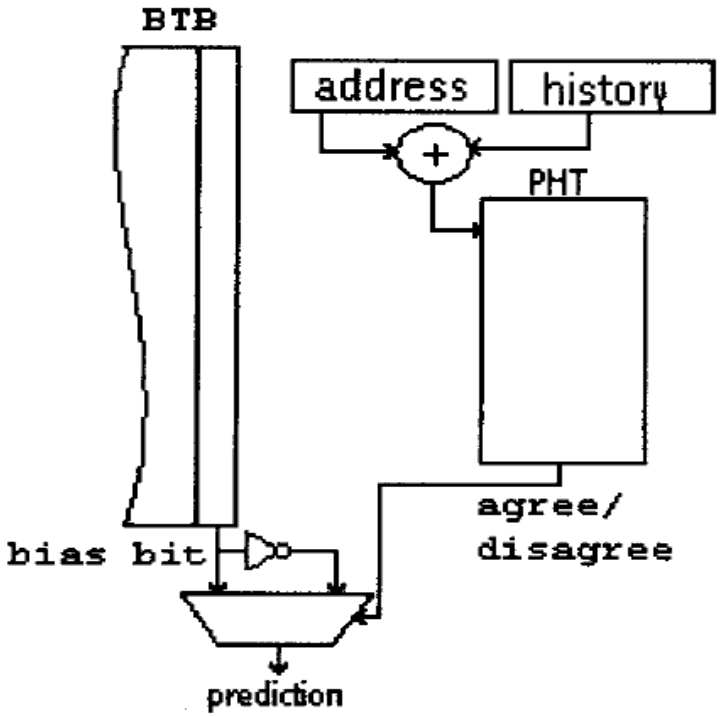
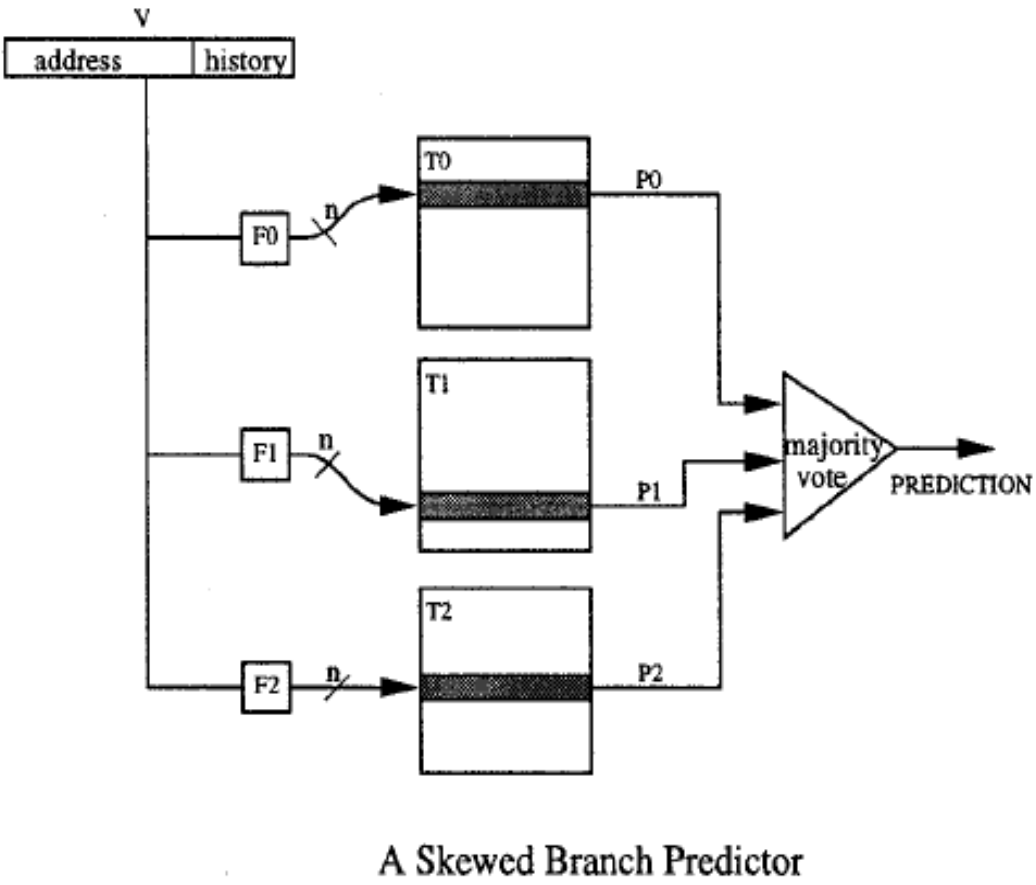


Fig 3.3

Predictor Skewed [4][5][6]

El predictor skewed es un predictor dinámico de dos niveles, que parte de la afirmación de que la mayoría de los casos de aliasing no se deben a que el tamaño de la PHT sea escaso, sino a la falta de asociatividad en ella (es decir, varias instrucciones de salto indexan a la misma entrada de la PHT, produciendo un conflicto). La mejor forma de combatir estos conflictos es introduciendo la asociatividad por conjuntos en la PHT, pero ésta no es una buena solución en términos del coste, y por tanto no es viable. En su lugar, esta estrategia de predicción emula la asociatividad utilizando una funciones especiales de sesgo.

Este pedictor divide la PHT en tres bancos del mismo tamaño, y utiliza una función hash distinta en cada banco para indexar a un contador saturado por cada banco (*f1*, *f2* y *f3*), como podemos ver en el esquema más abajo. La predicción final es dada de acuerdo al voto mayoritario entre los 3 bancos haciendo uso de las ideas de un Tournament Predictor. Es necesario asegurar que si un salto compite con otro por el mismo contador en un banco, no será así en ninguno de los otros dos bancos, y el voto mayoritario producirá una predicción libre de aliasing.



A Skewed Branch Predictor

Fig 3.4

Predictor Bi-mode [10]

El predictor de saltos Bi-mode surge con el objetivo de reducir drásticamente el problema del aliasing destructivo en esquemas indexados por la historia global de saltos. Esta técnica, divide la tabla de contadores saturados de 2º nivel ó PHT en dos mitades. Dado un patrón de historia determinado, se seleccionan dos contadores, uno de cada mitad, según el esquema mostrado más abajo (llamaremos a ambas mitades *predictores de dirección*). Además, tendremos otra tabla de contadores de 2 bit, indexada únicamente por la dirección de la instrucción de salto, que se utiliza para devolver una predicción final de entre los predictores de dirección. Esta tabla la llamaremos *predictor de elección*. Por tanto, la predicción final resultará del estado del contador seleccionado de entre los predictores de dirección.

Este esquema minimiza el coste hardware del predictor Yags al no contar con un tag de acceso, convirtiendo las cachés en simples tablas de contadores saturados.

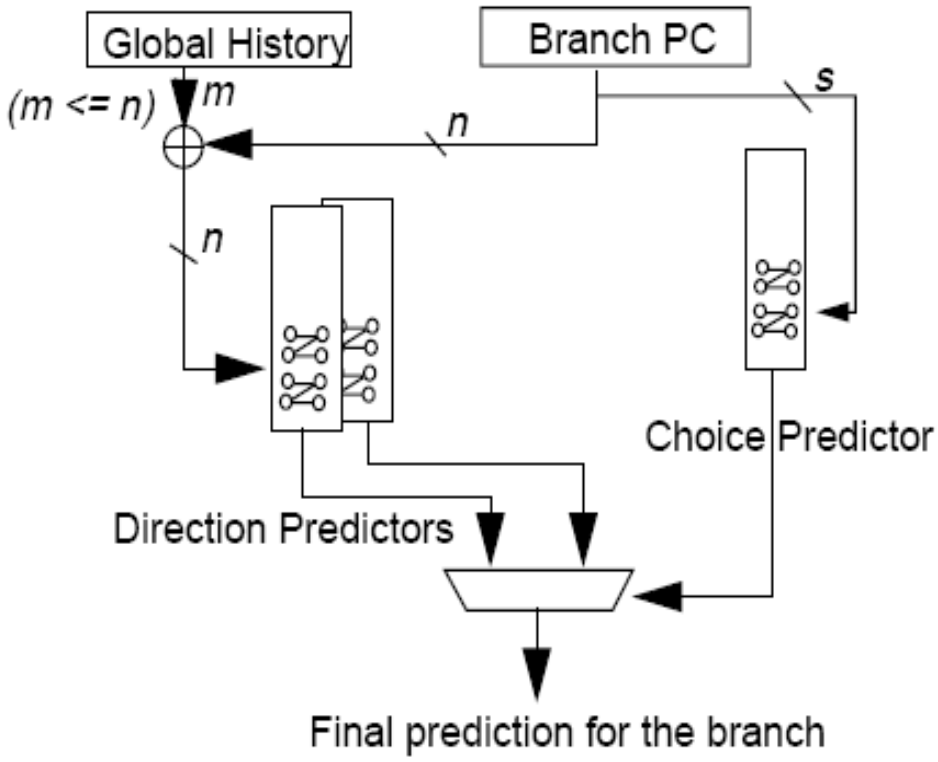


Fig 3.5

Predictor YAGS [8]

El predictor de saltos YAGS hace uso de los mejores aspectos de algunos de los predictores descritos anteriormente, especialmente el Bi-Mode, para intentar conseguir un mejor rendimiento.

Se observa que para cada salto necesitamos almacenar dos cosas:

- su comportamiento la PHT.
- los saltos que no están de acuerdo con la respuesta ofrecida por la PHT.

Utilizamos una tabla de contadores saturados que elegirá entre una de las dos cachés: taken caché que guarda los saltos tomados y not taken caché que guarda saltos no tomados; ambas están indexadas mediante un tag de entre 6 y 8 bits. En caso de que no se encuentre en ninguna de las cachés usará su propia predicción. La idea de este predictor es guardar los saltos que no “respetan” el comportamiento habitual marcado en la PHT; de esta forma podemos saber si un salto es un caso especial y obtener la predicción en las cachés.

Este esquema de predicción elimina de forma virtual todo el aliasing de los programas habituales(single thread). También obtiene un buen rendimiento en aplicaciones multi-threading ya que no suele haber mucha interferencia en la historia de los saltos entre hilos.

Todo este rendimiento bruto no sale gratis: el acceso a las cachés marcadas con tags es lento y necesita más potencia que otros esquemas más sencillos. Además el coste hardware de las estructuras necesarias para la implementación de este predictor es elevado.

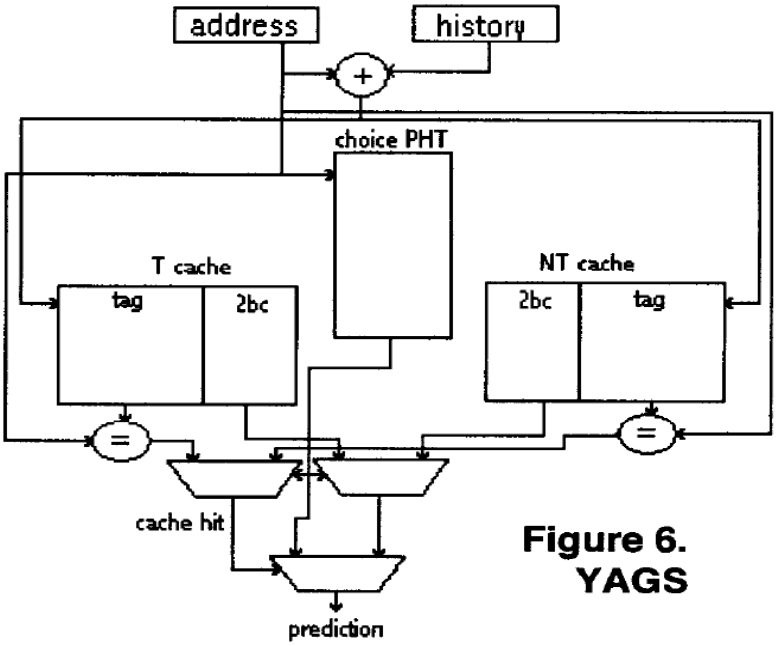


Fig 3.6

Predictor Perœptrón [11]

Aplicando conocimientos de inteligencia artificial, este predictor utiliza la teoría del perceptrón de una capa para tratar de aprender el comportamiento de un salto. Concretamente trata de aprender la correlación del resultado de un salto y el registro de historia.

Estas correlaciones son representadas por un perceptrón. Por cada salto se actualizan los valores de los pesos de perceptrón de la entrada correspondiente, así si un salto “insiste” en mostrar un determinado comportamiento, los valores de estos pesos crecerán o decrecerán de forma que un puntual salto con un resultado distinto al marcado por el perceptrón no logrará hacer que cambie la dirección del salto y por tanto no resultarán contaminadas las predicciones.

Este proceso se conoce en inteligencia artificial como aprendizaje e irá definiendo la correlación con el registro de historia de un determinado salto de forma que a cada aparición estará más cerca de conseguir la predicción correcta o se reafirmará en ella si ya era la adecuada.

La estructura de este predictor se puede representar como una tabla de perceptrones, cada uno representado como un vector con tantos bits como el ancho del registro de historia. Esta tabla es accedida utilizando una función de dispersión para minimizar el efecto del aliasing.

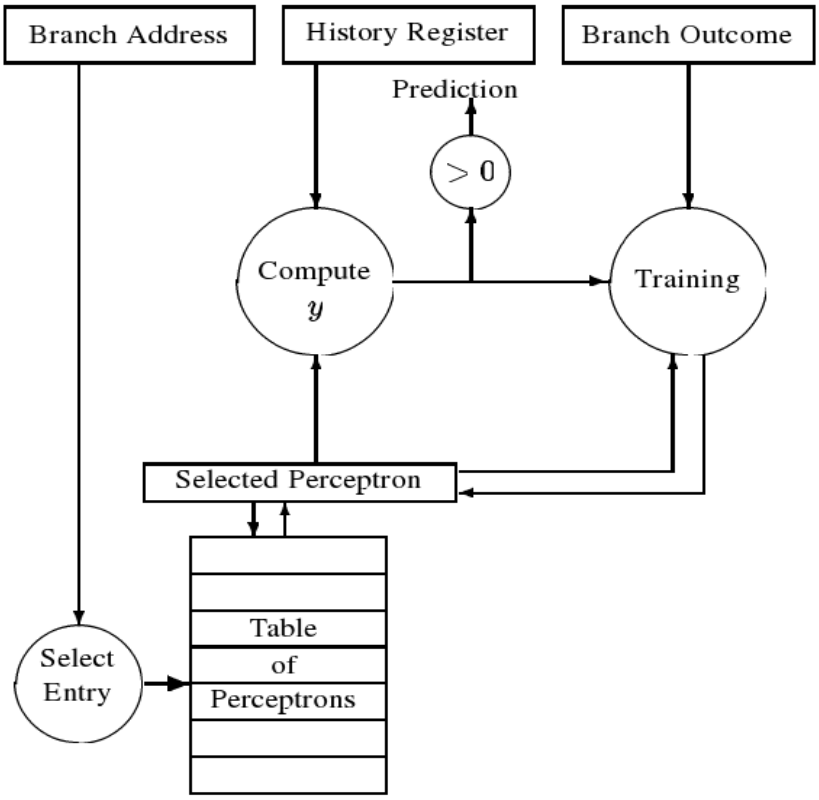

Branch Address
History Register
Branch Outcome
Prediction
> 0
Compute
y
Training
Selected Perceptron
Table
of
Perceptrons
Select
Entry

Fig 3.7

4. Entorno Experimental

La enorme importancia del rendimiento como parámetro a la hora de diseñar un microprocesador ha convertido al *benchmarking* en un punto clave en el proceso de diseño. Una gran variedad de benchmarks se han propuesto, incluyendo LINPACK, Whetstone, MediaBench, y muchos otros, aunque el conjunto de benchmarks más conocido y utilizado en los últimos tiempos es SPEC (*Standard Performance Evaluation Corporation*), y ha ejercido gran influencia en el diseño de micro-procesadores de propósito general.

El dominio de aplicaciones para sistemas empotrados es el segmento de más rápido crecimiento de mercado en la industria de microprocesadores. El amplio abanico de aplicaciones para sistemas empotrados hace muy difícil definir un perfil común de carga de trabajo. El conjunto de aplicaciones puede ir desde sistemas sensores en simples microcontroladores hasta complejos smartphones con la funcionalidad de PCs de escritorio, que además llevan soporte para comunicaciones inalámbricas.

Para realizar las pruebas hemos utilizado un paquete de benchmarks llamado MiBench[3]. En esta suite, se presenta un conjunto de 35 aplicaciones para sistemas empotrados, divididos en 6 categorías, cada una de ellas representando un área del mercado. Todos los programas se presentan en código C estándar, y son portables a cuaqluier plataforma con soporte para compiladores.

Las 6 categorías son:

- *Control Automático e Industrial*. Representa la integración de procesadores empotrados en los sistemas de control, como pueden ser controladores de airbag, centralitas electrónicas de motores, y sistemas sensores. Los programas utilizados para caracterizar estas situaciones son un test matématico (basicmath), un contador de bits (bitcount), un algoritmo de ordenación (qsort) y un programa de reconocimiento de contornos (susan).
- *Redes*. Representa el uso de este tipo de procesadores en dispositivos de red como routers, switches, y gateways. Los programas que caracterizan estas situaciones son un algoritmo de caminos mínimos en grafos (dijkstra), y búsquedas/inserciones/borrado de datos en árboles, simulando tablas de rutado (patricia).
- *Seguridad de datos*. Este grupo de aplicaciones incluye algunos algoritmos de encriptación/desencriptación de datos, y de hashing (blowfish, pgp, rijndael, sha).
- *Dispositivos de consumo*. Representa la amplia variedad de dispositivos que han ido apareciendo y ganando popularidad, como pueden ser escáneres, cámaras digitales, PDAs, reproductores MP3, etc. Este grupo incluye aplicaciones de tratamiento de imágenes (jpeg, tiff2bw, tiff2rgba, tiffdither, tiffmedian), y de tratamiento de señales de audio (lame, mad).
- *Ofimática*. Incluye diversas aplicaciones de procesamiento de textos, de impresión, fax (ghostscript, ispell, rsynth, sphynx, stringsearch).
- *Telecomunicaciones.* Representa el sector del mercado probablemente con más crecimiento, como es el de las telecomunicaciones, con Internet y las

comunicaciones inalámbricas como abanderados. Incluye aplicaciones de encoding/decoding de voz, checksums, y análisis de frecuencia (CRC32, FFT, ADPCM, GSM...).

Debido a la naturaleza particular de los procesadores ARM y los dispositivos en los que éstos se utilizan, hemos realizado las pruebas para tamaños de hardware reducidos que comprenderán desde 0.064KB hasta 2KB doblando el hardware utilizado cada vez y ajustando los diferentes parámetros de cada predictor a sus óptimos experimentales para el tamaño seleccionado. Para conseguir estos valores óptimos, dentro de las peculiaridades de cada uno y ayudándonos de resultados experimentales de otros estudios sobre procesadores de propósito general, hemos lanzado una gran cantidad de diferentes configuraciones de las que hemos seleccionado las mejores. Además, en los resultados empíricos que hemos ido obteniendo constatamos que tamaños superiores a 2KB no suponían una mejora suficiente a tener en cuenta y no solo eso sino que los predictores iban llegando a su punto de saturación según se acercaban a los 8KB. Un último detalle a tener en cuenta es que no se ha variado el tamaño de la BTB, fijado en los valores por defecto del simulador-ARM de 512 entradas y una asociatividad de 4 vias.

Las configuraciones concretas utilizadas son las siguientes:

TAMAÑO: 2KB

Configuración:

	Bimodal	Gshare	Agree	Yags	Percept.	Skewed	Bimode
L1	8192	1	1	1	1	-	1
L2/PHT	-	8192	8192	-	1024	-	-
Choice PHT/perceptrones	-	-	-	2048	-	-	4096
dir PHT/cachés	-	-	-	2x1024*	-	-	2x2048
bancos	-	-	-	-	-	**	-
historia	-	13	16	10	16	16	16

(tabla 1)

* Yags: cachés asociativas de 1 vía. 6 bit de Tag. El tamaño según la tabla es de 2,5KB por los que se ha ponderado su valor usando el predictor de 1,5KB con igual PHT y cachés de 512 entradas.

** Skewed: los resultados se han tomado como la media ponderada entre el predictor de 1.5KB con bancos de 2048 entradas y el de 3KB con bancos de 4096 entradas.

TAMAÑO: 1KB

Configuración:

	Bimodal	Gshare	Agree	Yags	Percept.	Skewed	Bimode
L1	4096	1	1	1	1	-	1
L2/PHT	-	4096	4096	-	1024	-	-
Choice PHT/perceptrones	-	-	-	2048	-	-	2048
dir PHT/cachés	-	-	-	2x512*	-	-	2x1024
bancos	-	-	-	-	-	**	-
historia	-	12	16	9	8	16	16

(tabla 2)

* Yags: cachés asociativas de 1 vía. 6 bit de Tag. El tamaño según la tabla es de 1,5KB por los que se ha ponderado su valor usando el predictor de 0,75KB con igual PHT y cachés de 256 entradas.

** Skewed: los resultados se han tomado como la media ponderada entre el predictor de 1.5KB con bancos de 2048 entradas y el de 0,768KB con bancos de 1024 entradas.

TAMAÑO: 0.5KB

Configuración:

	Bimodal	Gshare	Agree	Yags	Percept.	Skewed	Bimode
L1	2048	1	1	1	1	-	1
L2/PHT	-	2048	2048	-	512	-	-
Choice PHT/perceptrones	-	-	-	1024	-	-	1024
dir PHT/cachés	-	-	-	2x256*	-	-	2x512
bancos	-	-	-	-	-	*	-
historia	-	11	16	8	8	16	16

(tabla 3)

* Skewed: los resultados se han tomado como la media ponderada entre el predictor de 0,768KB con bancos de 1024 entradas y el de 0,384KB con bancos de 512 entradas.

TAMAÑO: 0.25KB

Configuración:

	Bimodal	Gshare	Agree	Yags	Percept.	Skewed	Bimode
L1	1024	1	1	1	1	-	1
L2/PHT	-	1024	1024	-	256	-	-
Choice PHT/perceptrones	-	-	-	256	-	-	512
dir PHT/cachés	-	-	-	2x128*	-	-	2x256
bancos	-	-	-	-	-	**	-
historia	-	10	16	7	8	16	16

(tabla 4)

* Yags: cachés asociativas de 1 vía. 6 bit de Tag. El tamaño según la tabla es de 0,318KB por los que se ha ponderado su valor usando el predictor de 0,187KB con igual PHT y cachés de 128 entradas.

** Skewed: los resultados se han tomado como la media ponderada entre el predictor de 0,384 KB con bancos de 512 entradas y el de 0,192KB con bancos de 256 entradas.

TAMAÑO: 0.125KB

Configuración:

	Bimodal	Gshare	Agree	Yags	Percept.	Skewed	Bimode(***)
L1	512	1	1	1	1	-	1
L2/PHT	-	512	512	-	256	-	-
Choice PHT/perceptrones	-	-	-	128	-	-	512-256
dir PHT/cachés	-	-	-	2x64*	-	-	2x128-64
bancos	-	-	-	-	-	**	-
historia	-	9	16	6	4	16	16

(tabla 5)

* Yags: cachés asociativas de 1 vía. 6 bit de Tag. El tamaño según la tabla es de 0,16KB por los que se ha ponderado su valor usando el predictor de 0,096KB con igual PHT y cachés de 32 entradas.

** Skewed: los resultados se han tomado como la media ponderada entre el predictor de 0,192 KB con bancos de 256 entradas y el de 0,096KB con bancos de 128 entradas.

*** Bimode: Se ha tomado una media ponderada entre el predictor de 0,189KB con PHT de 512 entradas y dir PHT's de 128 entradas y, el predictor de 0,096KB con PHT de 256 entradas y dir PHT's de 64 entradas.

TAMAÑO: 0.064KB

Configuración:

	Bimodal	Gshare	Agree	Yags	Percept.	Skewed	Bimode(***)
L1	256	1	1	1	1	-	1
L2/PHT	-	256	256	-	128	-	-
Choice PHT/perceptrones	-	-	-	128	-	-	128
dir PHT/cachés	-	-	-	2x16*	-	-	2x64
bancos	-	-	-	-	-	**	-
historia	-	8	16	5	4	16	16

(tabla 6)

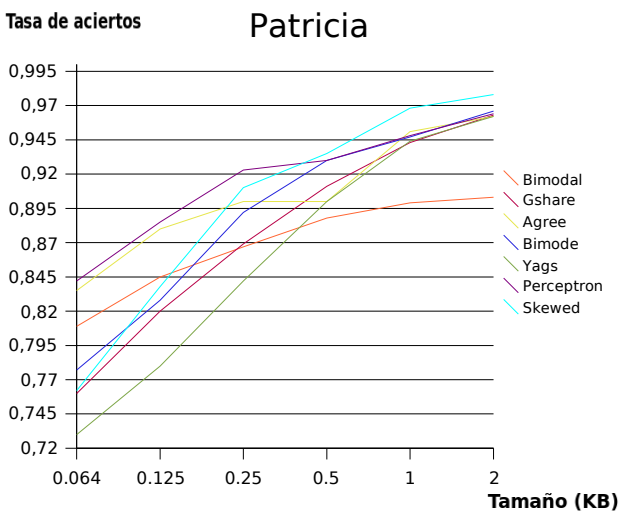
* Yags: cachés asociativas de 1 vía. 4 bit de Tag.

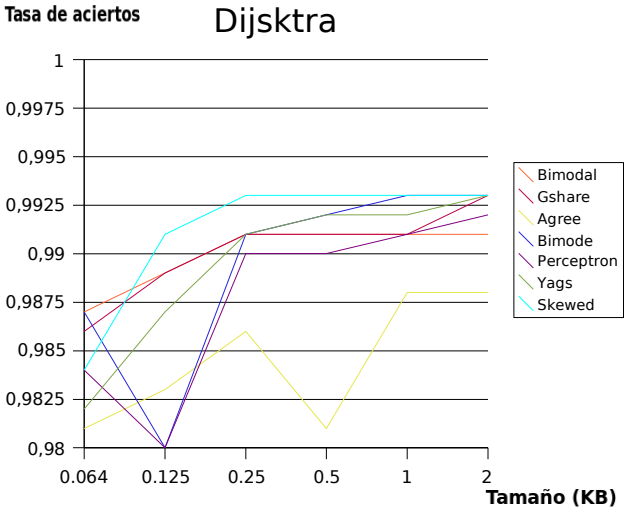
** Skewed: los resultados se han tomado como la media ponderada entre el predictor de 0,096KB con bancos de 128 entradas y el de 0,048KB con bancos de 64 entradas.

5. Resultados experimentales

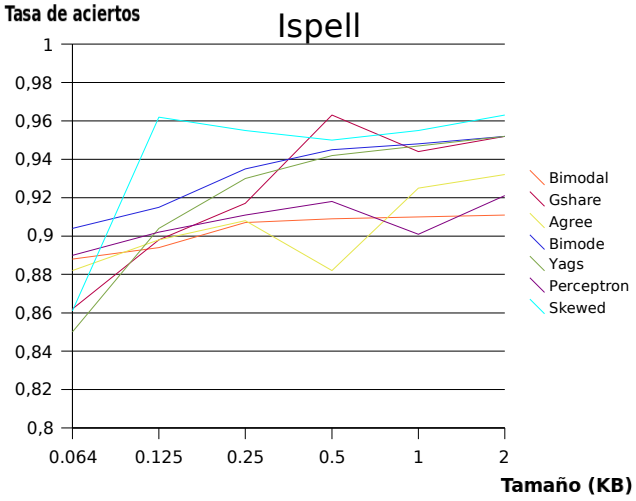
A continuación presentamos los resultados obtenidos en el simulador-ARM tras la ejecución de una amplia selección de los benchmarks que nos ofrece la suite MiBench. El primer conjunto de gráficas representa la evolución de la tasa de aciertos con respecto al tamaño de cada predictor particular.

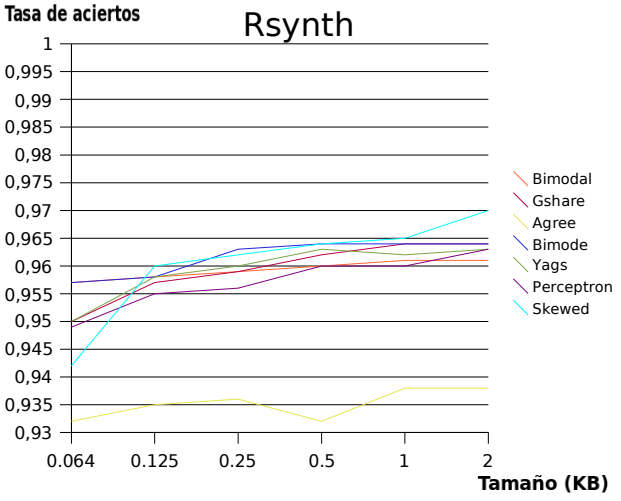
REDES:

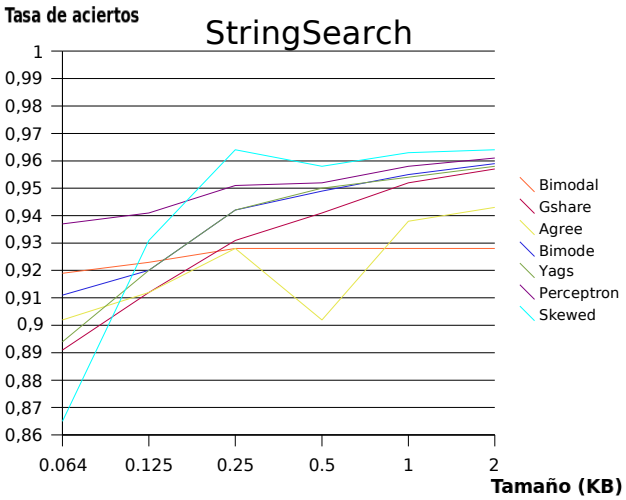




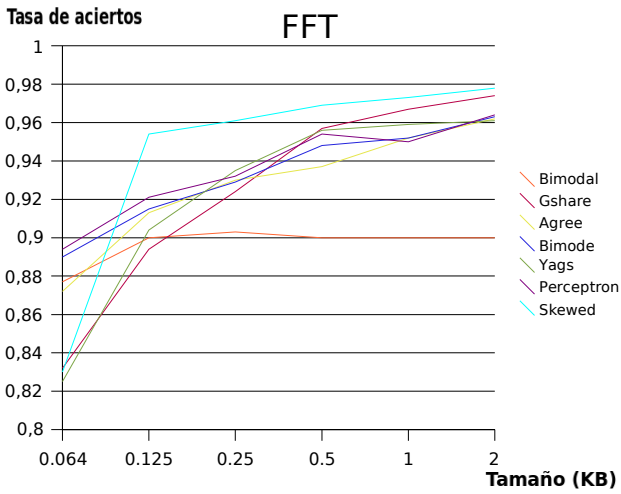
OFICINA:

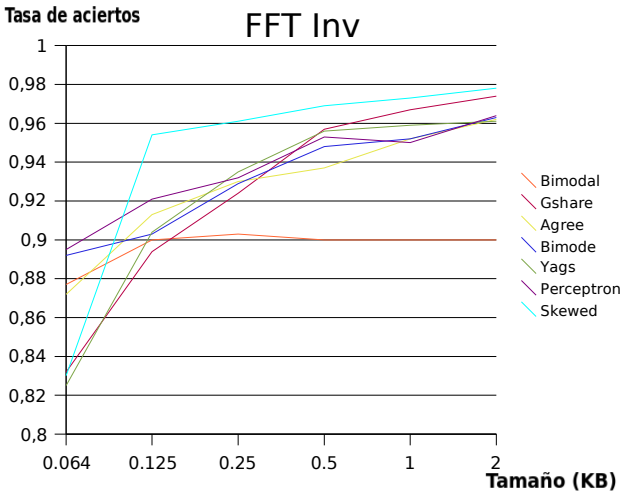


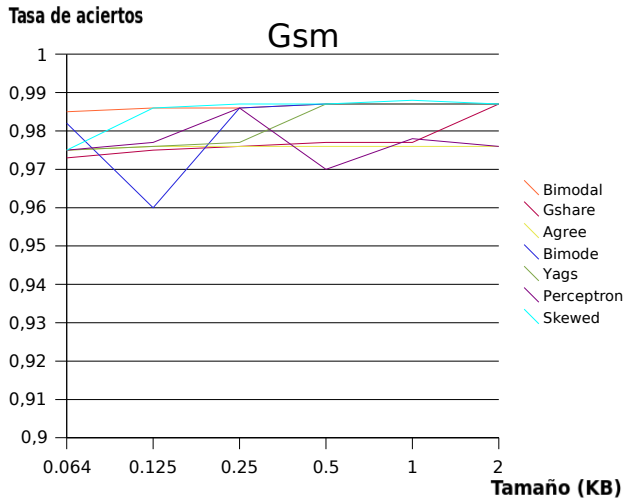




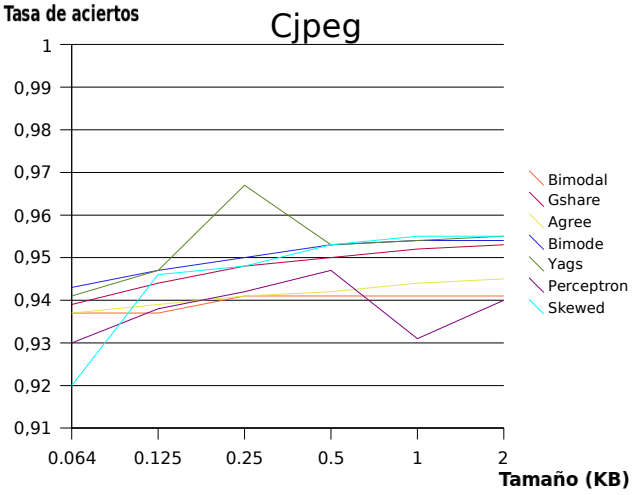
TELECOMUNICACIONES:

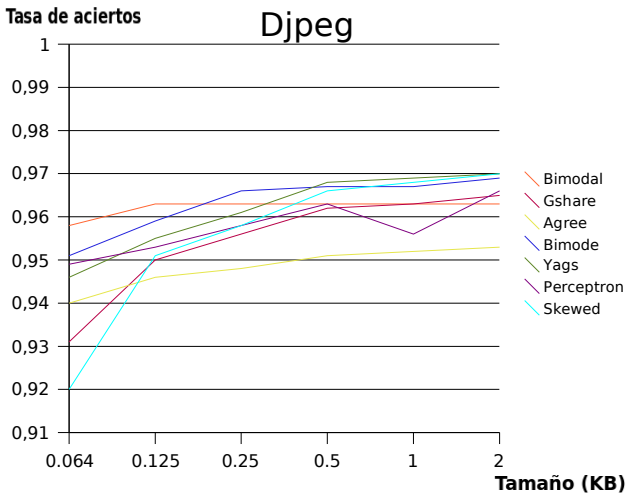


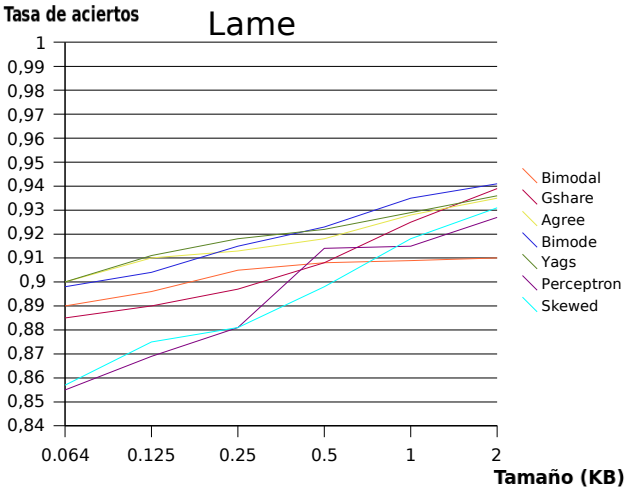


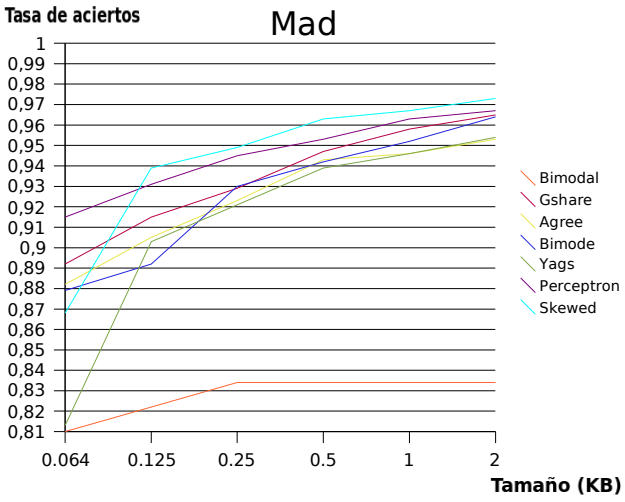


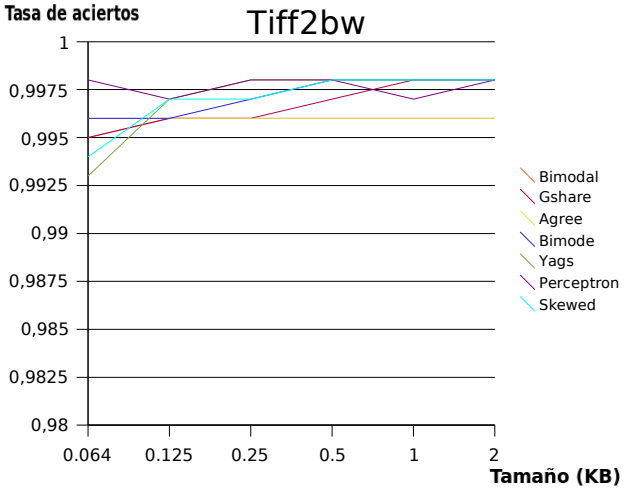
CONSUMO:

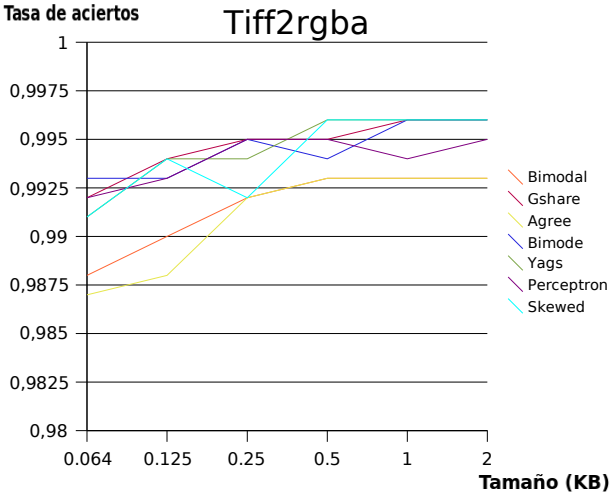

Cjpeg
Tasa de aciertos
1
0,99
0,98
0,97
0,96
0,95
0,94
0,93
0,92
0,91
0.064
0.125
0.25
0.5
1
2
Tamaño (KB)
Bimodal
Gshare
Agree
Bimode
Yags
Perceptron
Skewed

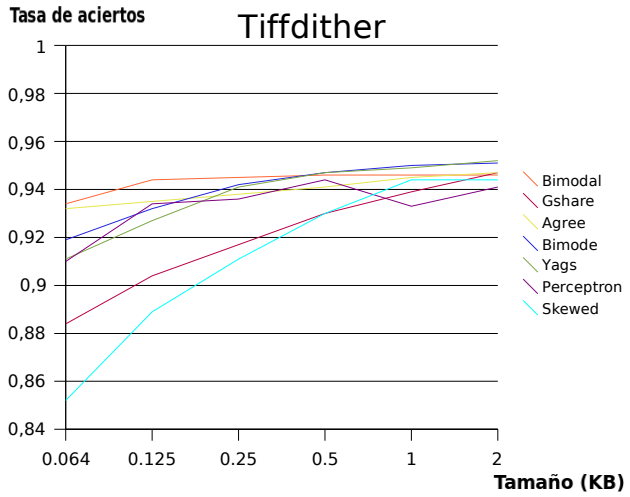

Djpeg
Tasa de aciertos
1
0,99
0,98
0,97
0,96
0,95
0,94
0,93
0,92
0,91
0.064
0.125
0.25
0.5
1
2
Tamaño (KB)
Bimodal
Gshare
Agree
Bimode
Yags
Perceptron
Skewed

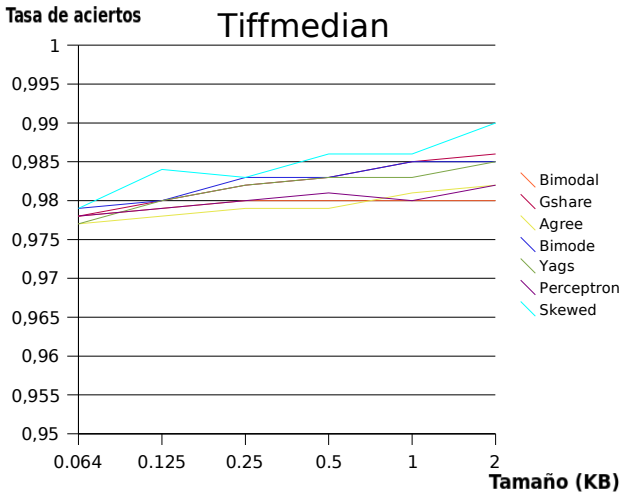

Lame
Tasa de aciertos
1
0,99
0,98
0,97
0,96
0,95
0,94
0,93
0,92
0,91
0,9
0,89
0,88
0,87
0,86
0,85
0,84
0.064
0.125
0.25
0.5
1
2
Tamaño (KB)
Bimodal
Gshare
Agree
Bimode
Yags
Perceptron
Skewed


Mad
Tasa de aciertos
1
0,99
0,98
0,97
0,96
0,95
0,94
0,93
0,92
0,91
0,9
0,89
0,88
0,87
0,86
0,85
0,84
0,83
0,82
0,81
0.064
0.125
0.25
0.5
1
2
Tamaño (KB)
Bimodal
Gshare
Agree
Bimode
Yags
Perceptron
Skewed

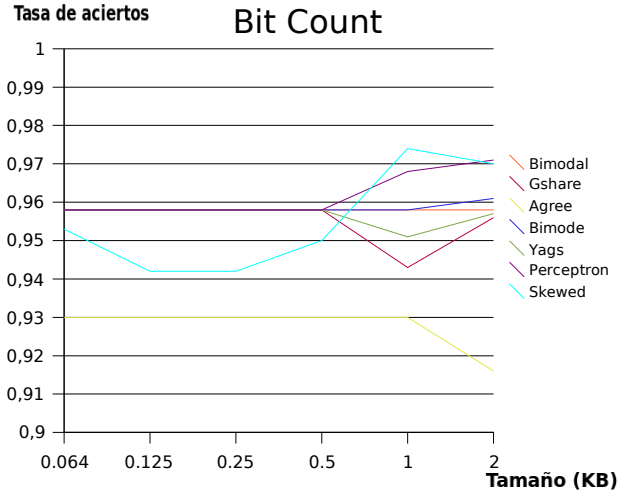

Tiff2bw
Tasa de aciertos
1
0,9975
0,995
0,9925
0,99
0,9875
0,985
0,9825
0,98
0.064
0.125
0.25
0.5
1
2
Tamaño (KB)
Bimodal
Gshare
Agree
Bimode
Yags
Perceptron
Skewed

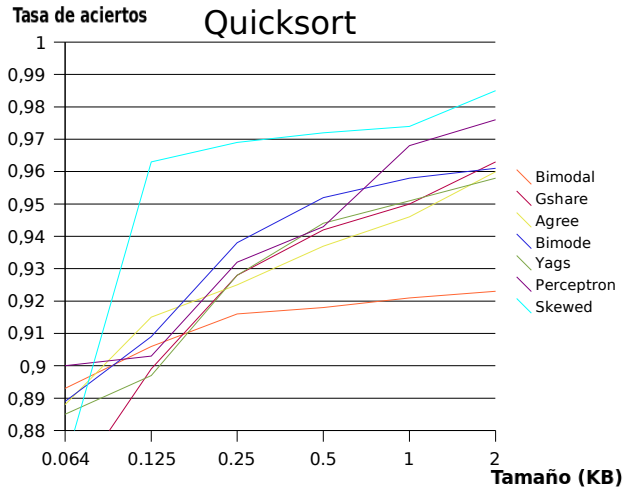

Tiff2rgba
Tasa de aciertos
1
0,9975
0,995
0,9925
0,99
0,9875
0,985
0,9825
0,98
0.064
0.125
0.25
0.5
1
2
Tamaño (KB)
Bimodal
Gshare
Agree
Bimode
Yags
Perceptron
Skewed



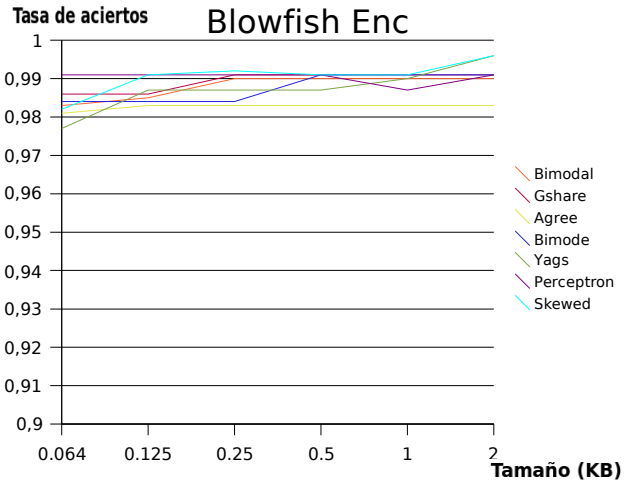


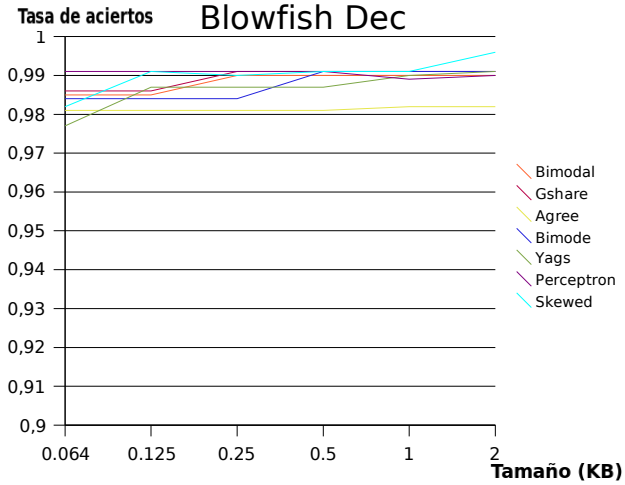
CONTROL AUTOMÁTICO:

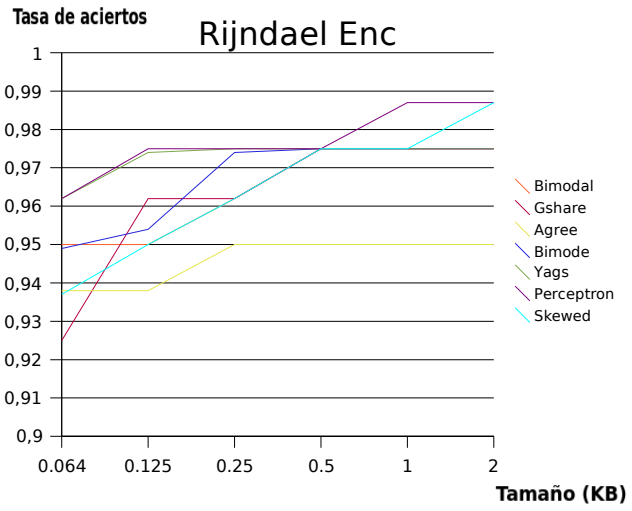


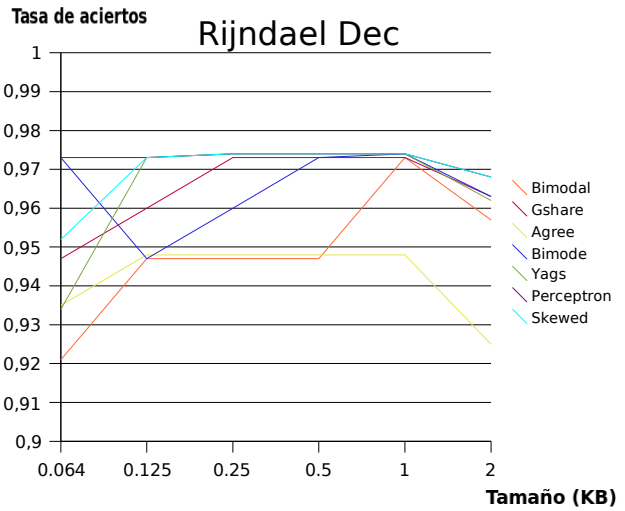


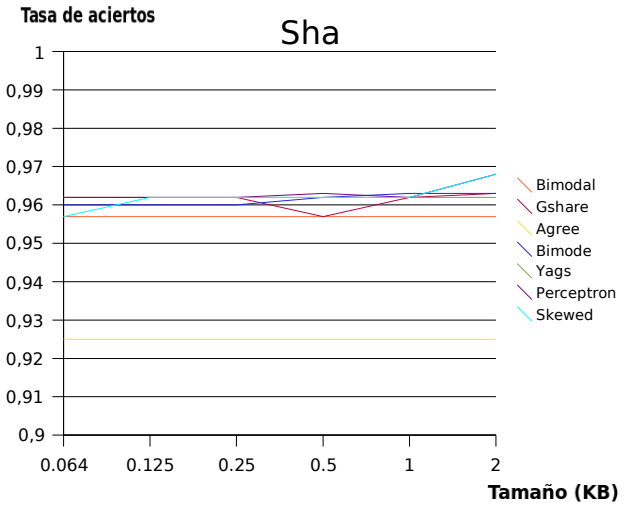
SEGURIDAD:











Se observa una importante dependencia respecto de la naturaleza de cada benchmark.

1. Redes: el comportamiento de los predictores en esta gama de benchmarks está muy repartido y no se observan grandes diferencias. En el patricia podemos destacar la rápida caída de la tasa según se reduce el tamaño de los predictores, debido a la interferencia de unos saltos con otros en las tablas de predicción al no disponer de suficiente espacio.

2. Oficina: se observa en general que son los que se desmarcan ligeramente de los demás presentando un rendimiento inferior debido a que se trata de programas de búsqueda y comparación de patrones de cadenas y caracteres que contienen muchas más instrucciones de condición. Las tasas llegan a caer por debajo del 90% de acierto y no superan apenas el 97%, se colocan de media en torno al 93% de aciertos.

3. Telecomunicaciones: muestran buenas tasas de acierto ya que los algoritmos tales como el gsm basado en cálculos gaussianos y las transformadas de Fourier se realizan de forma muy optimizada con procedimientos iterativos que no presentan muchas instrucciones de condición.

4. Consumo: son programas de tratamiento de imagen y audio que manejan grandes matrices de bits de forma extremadamente repetitiva tanto para transformaciones de sistemas de color, como para operaciones de codificación de audio, por lo que también muestran tasas altas. Presenta un comportamiento bastante estable sin acusar demasiado la reducción del tamaño salvo en los más reducidos.

5. Control automático: se podría decir que representa el caso extremo con el que generalizamos la conclusión posterior sobre el comportamiento de los predictores; el skewed presenta una tasa muy alta que se separa de un conjunto central muy igualado donde se encuentran todos excepto el bimodal, que es el que cae a lo más bajo de la tabla.

6. Seguridad: presentan un perfil de bucles largos para el tratamiento de grandes matrices por lo que el comportamiento de los predictores es bastante bueno llegando a tasas de entre un 96% a un 99%.

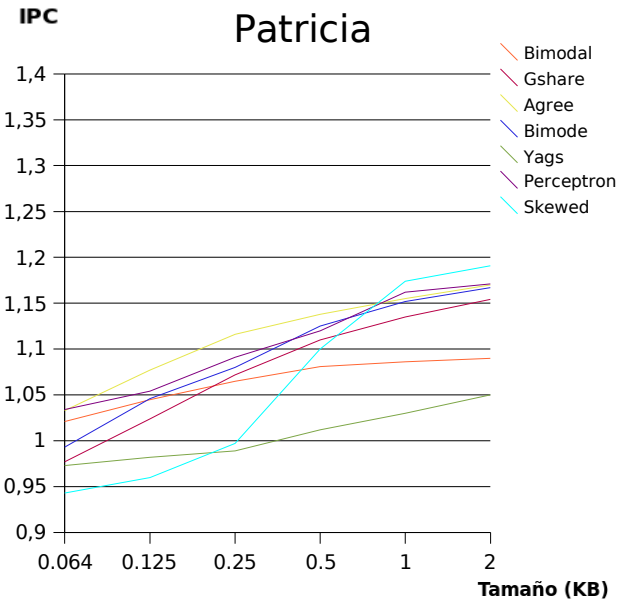
Si pasamos a comparar el comportamiento de los distintos predictores para todas las aplicaciones en general, observamos que el predictor skewed es el que mantiene una tasa superior de acierto, la división de la PHT en tres partes y la votación por mayoría entre las tres predicciones que se obtienen muestran resultados muy satisfactorios a la hora de reducir el aliasing. Por el contrario se observan las deficiencias del predictor bimodal, quedan bastante por debajo del resto aunque no obstante, y debido a su simplicidad, su relación coste hardware/rendimiento es alta.

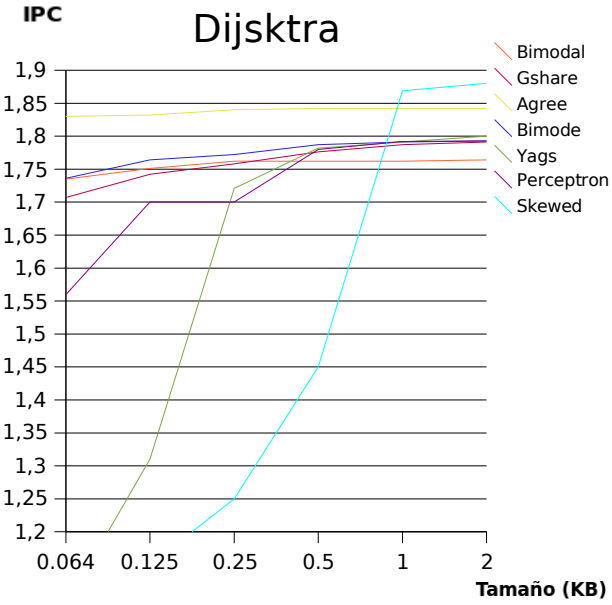
El predictor agree tampoco ha resultado muy eficaz, la predicción basada en el comportamiento inicial penaliza demasiado cuando un salto no presenta el comportamiento con el que entró en la BTB y más aún se produce una contaminación de la PHT que provoca demasiados fallos de predicción.

El resto de los predictores presentan un comportamiento ligeramente superior al gshare que es uno de los esquemas clásicos que venía ya implementado en el simulador-ARM e intentábamos superar. El innovador predictor perceptrón es el segundo mejor en comportamiento tras el skewed; este buen resultado demuestra la correlación existente entre los saltos y los bits de historia.

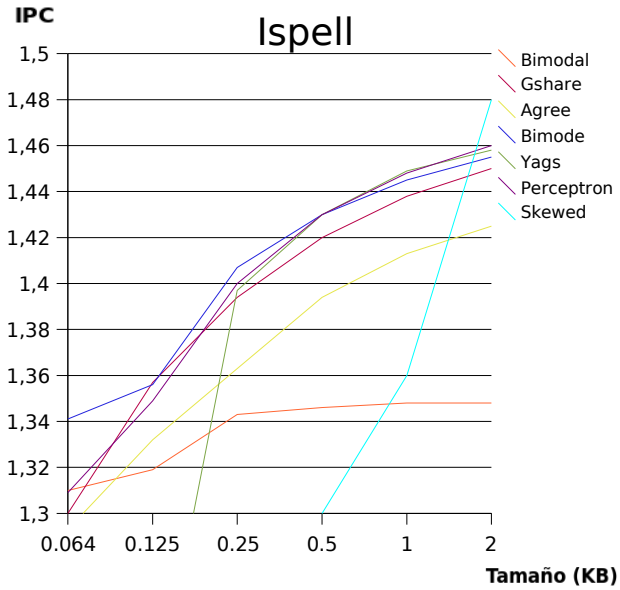
A continuación mostramos los resultados en cuanto al rendimiento obtenido. Utilizaremos una de las muchas medidas de rendimiento, las instrucciones por ciclo (IPC).

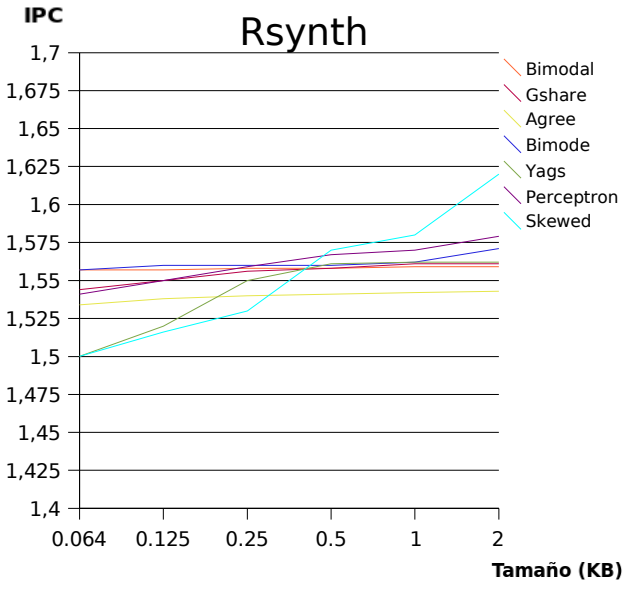
REDES:

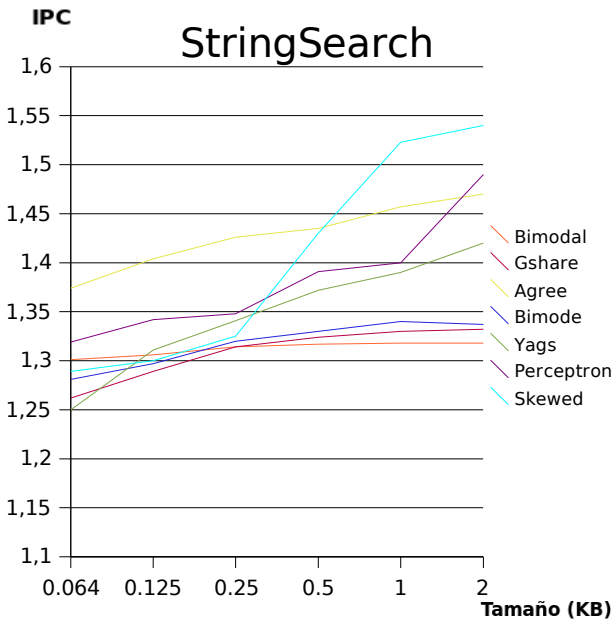




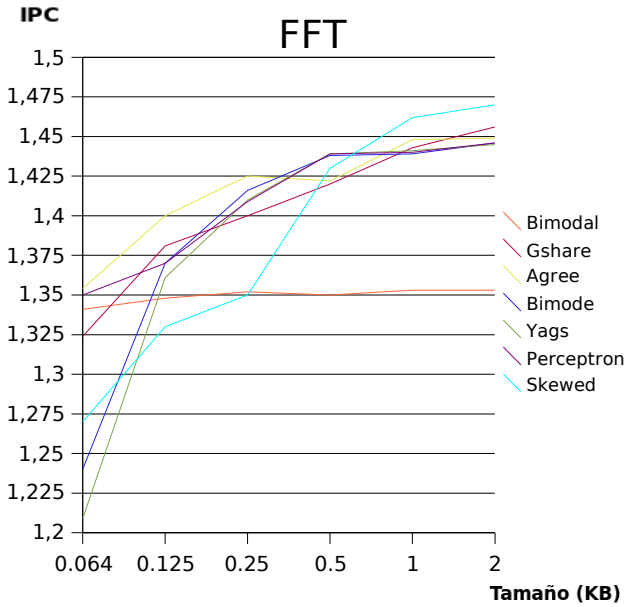
OFICINA:

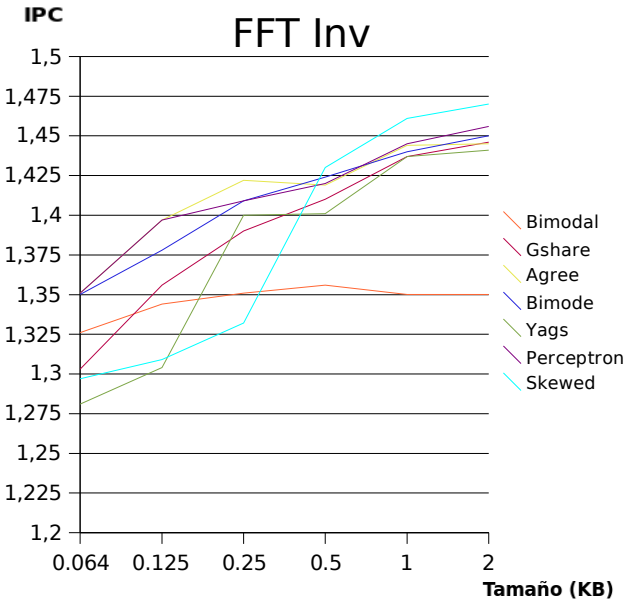


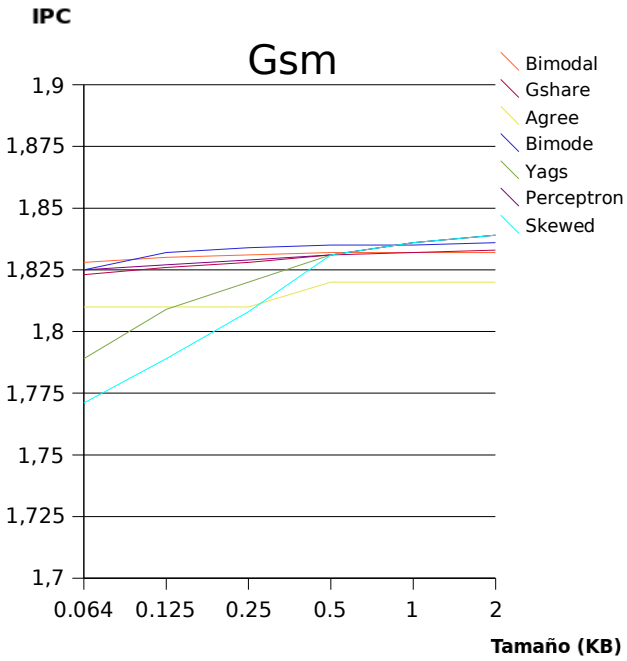




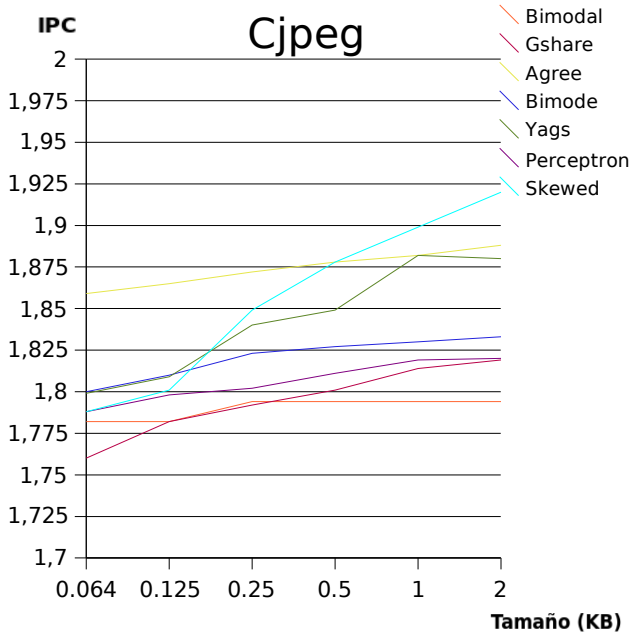
TELECOMUNICACIONES:

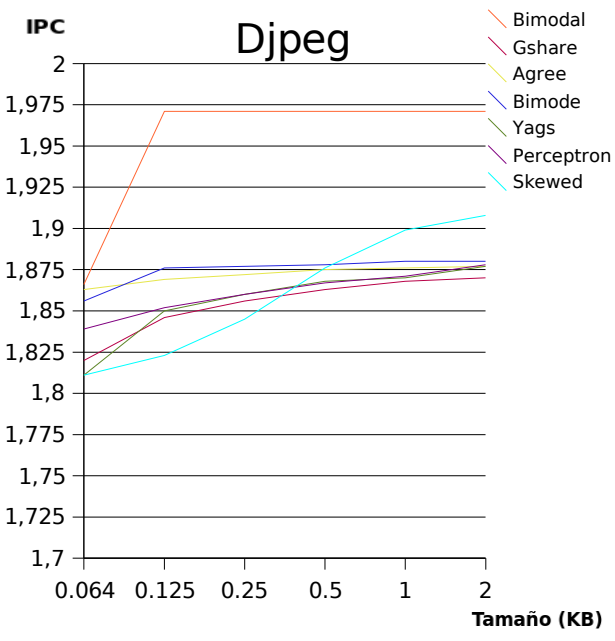


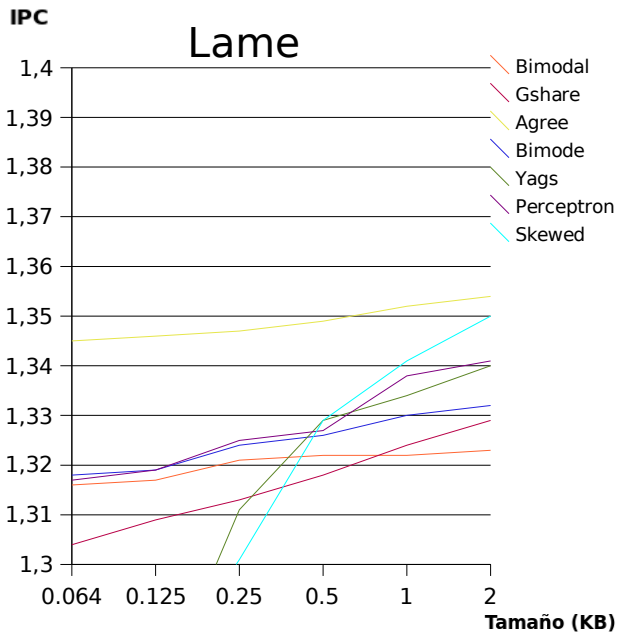


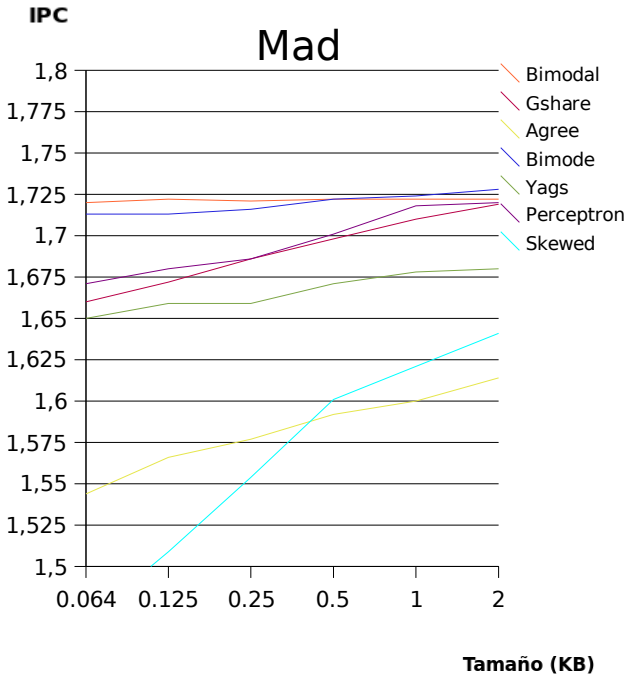

IPC
Gsm
1,9
1,875
1,85
1,825
1,8
1,775
1,75
1,725
1,7
0.064
0.125
0.25
0.5
1
2
Tamaño (KB)
Bimodal
Gshare
Agree
Bimode
Yags
Perceptron
Skewed

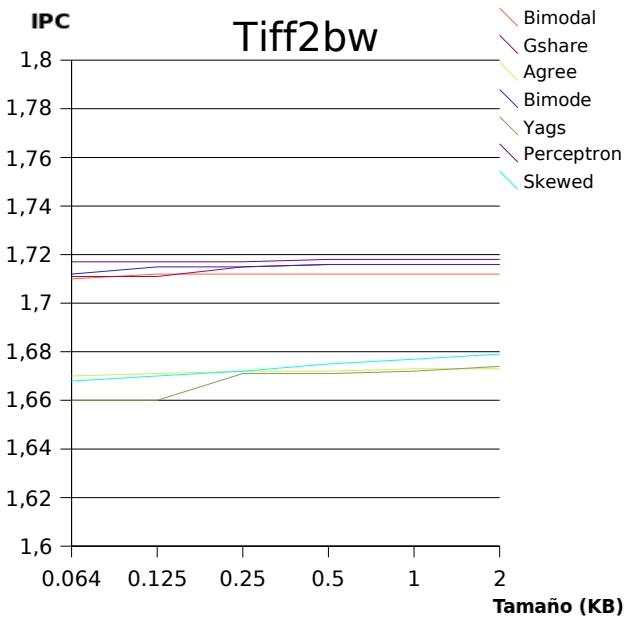
CONSUMO:

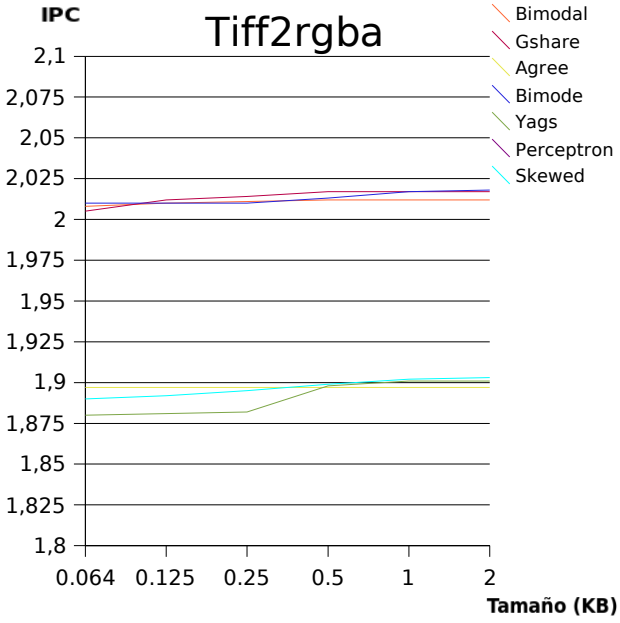

IPC
Cjpeg
2
1,975
1,95
1,925
1,9
1,875
1,85
1,825
1,8
1,775
1,75
1,725
1,7
0.064
0.125
0.25
0.5
1
2
Tamaño (KB)
Bimodal
Gshare
Agree
Bimode
Yags
Perceptron
Skewed

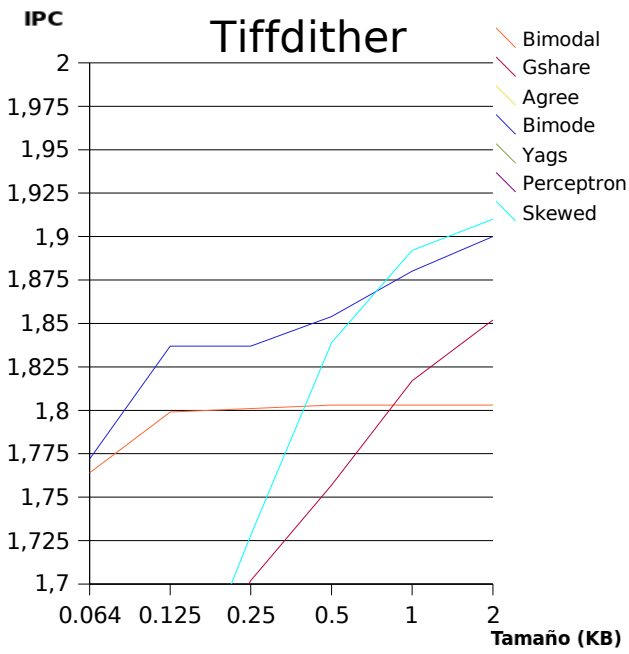

IPC
Djpeg
2
1,975
1,95
1,925
1,9
1,875
1,85
1,825
1,8
1,775
1,75
1,725
1,7
0.064
0.125
0.25
0.5
1
2
Tamaño (KB)
Bimodal
Gshare
Agree
Bimode
Yags
Perceptron
Skewed

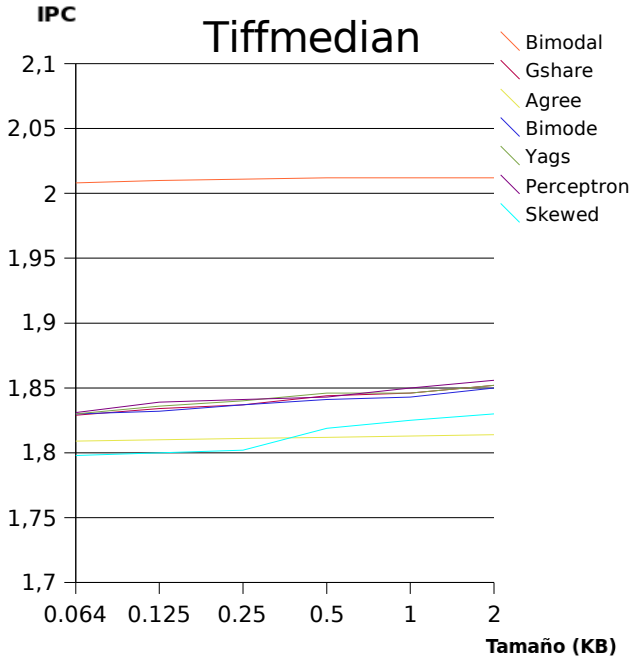

IPC
Lame
1,4
1,39
1,38
1,37
1,36
1,35
1,34
1,33
1,32
1,31
1,3
0.064
0.125
0.25
0.5
1
2
Tamaño (KB)
Bimodal
Gshare
Agree
Bimode
Yags
Perceptron
Skewed


IPC
Mad
1,8
1,775
1,75
1,725
1,7
1,675
1,65
1,625
1,6
1,575
1,55
1,525
1,5
0.064
0.125
0.25
0.5
1
2
Tamaño (KB)
Bimodal
Gshare
Agree
Bimode
Yags
Perceptron
Skewed

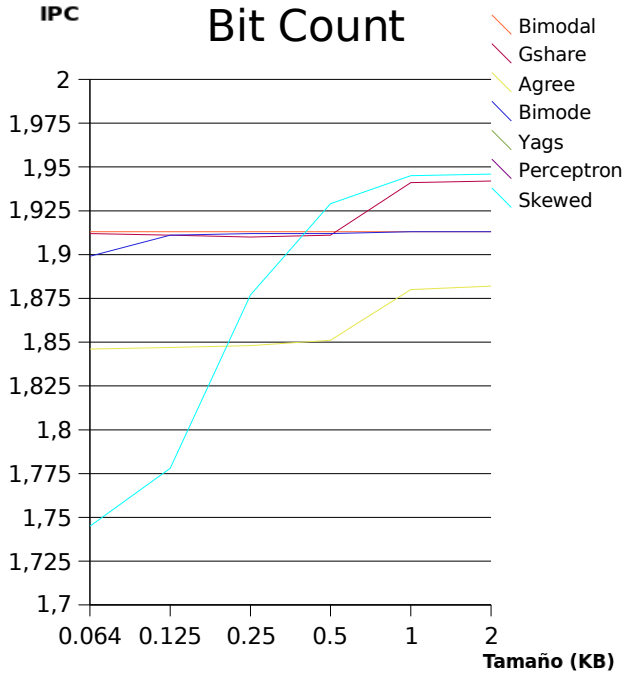

IPC
Tiff2bw
1,8
1,78
1,76
1,74
1,72
1,7
1,68
1,66
1,64
1,62
1,6
0.064
0.125
0.25
0.5
1
2
Tamaño (KB)
Bimodal
Gshare
Agree
Bimode
Yags
Perceptron
Skewed

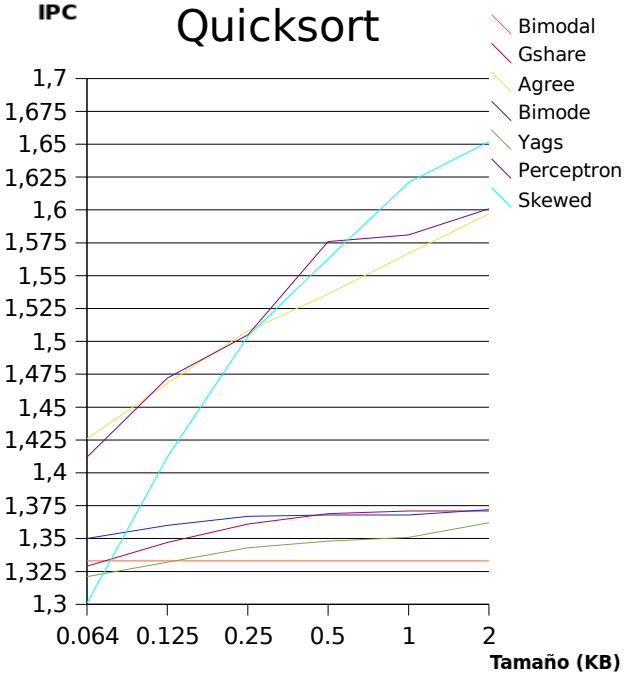

IPC
Tiff2rgba
2,1
2,075
2,05
2,025
2
1,975
1,95
1,925
1,9
1,875
1,85
1,825
1,8
0.064
0.125
0.25
0.5
1
2
Tamaño (KB)
Bimodal
Gshare
Agree
Bimode
Yags
Perceptron
Skewed



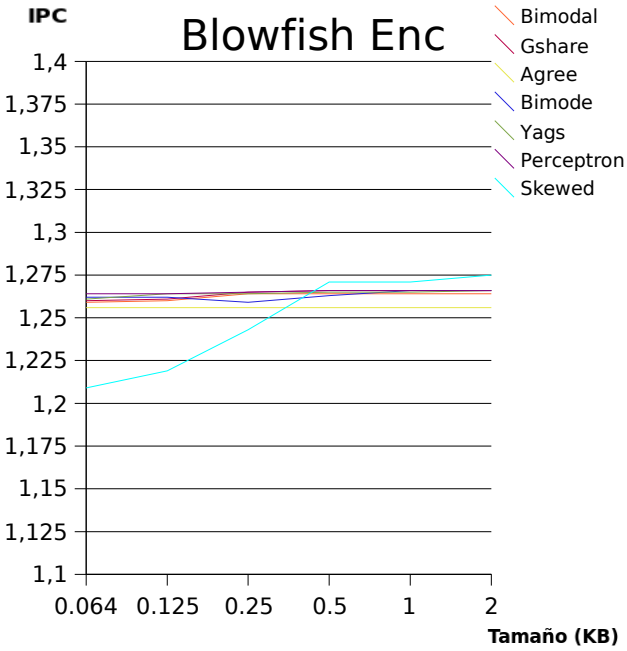


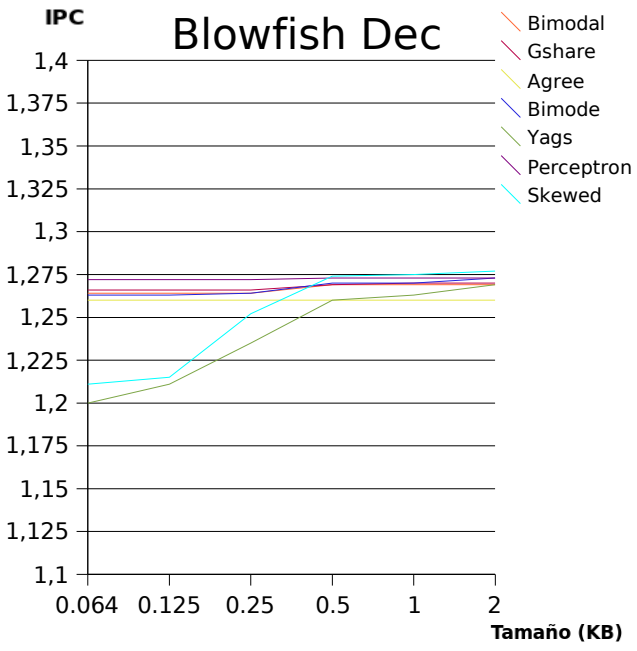
CONTROL AUTOMÁTICO:

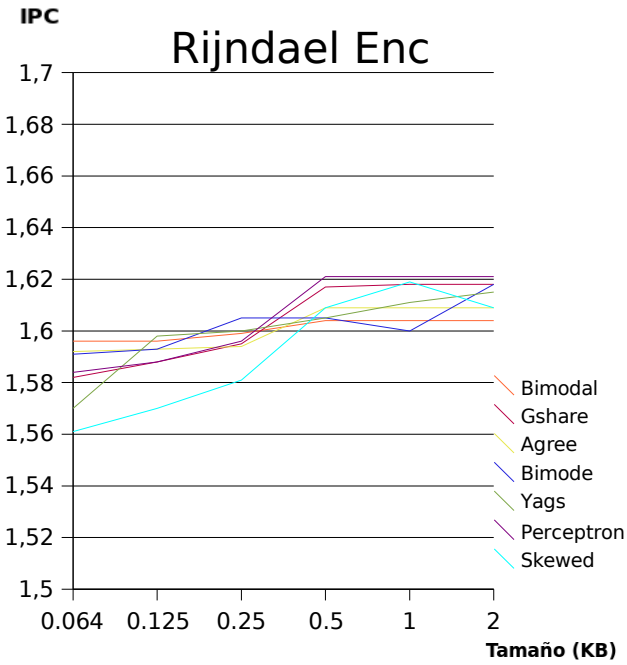


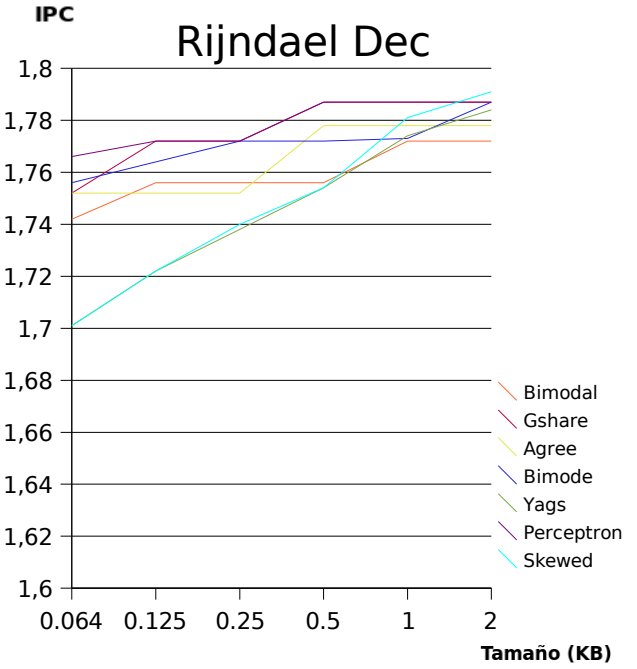


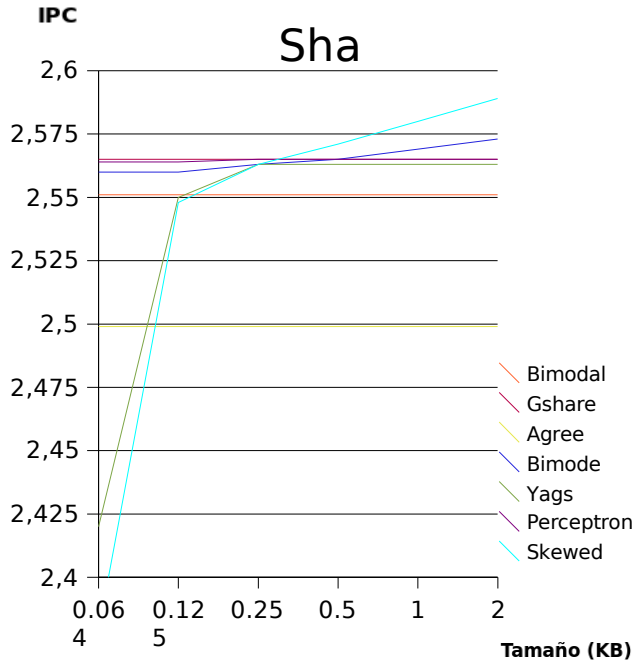
SEGURIDAD:


Blowfish Enc
IPC
1,4
1,375
1,35
1,325
1,3
1,275
1,25
1,225
1,2
1,175
1,15
1,125
1,1
0.064
0.125
0.25
0.5
1
2
Tamaño (KB)
Bimodal
Gshare
Agree
Bimode
Yags
Perceptron
Skewed


Blowfish Dec
IPC
1,4
1,375
1,35
1,325
1,3
1,275
1,25
1,225
1,2
1,175
1,15
1,125
1,1
0.064
0.125
0.25
0.5
1
2
Tamaño (KB)
Bimodal
Gshare
Agree
Bimode
Yags
Perceptron
Skewed


Rijndael Enc
IPC
1,7
1,68
1,66
1,64
1,62
1,6
1,58
1,56
1,54
1,52
1,5
0.064
0.125
0.25
0.5
1
2
Tamaño (KB)
Bimodal
Gshare
Agree
Bimode
Yags
Perceptron
Skewed


Rijndael Dec
IPC
1,8
1,78
1,76
1,74
1,72
1,7
1,68
1,66
1,64
1,62
1,6
0.064
0.125
0.25
0.5
1
2
Tamaño (KB)
Bimodal
Gshare
Agree
Bimode
Yags
Perceptron
Skewed



Podemos observar cómo, en general, los predictores muestran un comportamiento acorde con los resultados obtenidos anteriormente en la tasa de aciertos. Las diferencias se reducen debido a que ahora el rendimiento no sólo tiene en cuenta la parte del predictor de saltos sino que abarca toda la arquitectura del procesador.

El skewed alcanza un rendimiento superior al resto tal como era de esperar, llegando a superar al bimodal en más de 0.1 instrucción por ciclo de media y alcanzando una diferencia de 0.3 IPC en el conocido algoritmo de ordenación quicksort. El resto de predictores se mantienen agrupados en torno a valores similares.

Respecto al conjunto de benchmarks el peor rendimiento recae sobre los pertenecientes al grupo de oficina que se mantienen entre 1.2 y 1.6 IPC junto con los de telecomunicaciones. Cabe destacar el alto rendmiento que se oberva en el Sha, llegando el skewed a 2.585 intrucciones por ciclo.

6. Conclusiones y trabajos futuros

En este proyecto hemos evaluado una serie de técnicas complejas de predicción sobre un procesador empotrado. Concretamente, hemos empleado el ARM, un procesador ampliamente utilizado en sistemas empotrados comerciales. El resultado final demuestra que, si bien en algunos casos un predictor de salto agresivo apenas logra mejorar el rendimiento conseguido por un sencillo bimodal, en general tiene sentido utilizar estas técnicas de predicción tan agresivas, puesto que provocan una mejora sustancial en el tiempo de ejecución.

Si analizamos en detalle para los distintos predictores evaluados, observamos que el predictor de saltos Skewed es el que mantiene la mayor tasa de acierto, entre un 2% y un 6% respecto del resto, y un mejor rendimiento. También cabe destacar el predictor bimodal, que pese a ofrecer un rendimiento algo inferior, es un esquema a tener en cuenta debido a su bajo coste y simplicidad. Respecto del resto de esquemas, los más complejos como el Yags, perceptrón o Bimode no proporcionan una mejora tan grande como para invertir el hardware necesario, sobre todo en sistemas empotrados.

Existen varias líneas de investigación futura que podrían continuar el desarrollo de este proyecto. Una de ellas sería la prueba de este tipo de técnicas de predicción en otras arquitecturas empotradas distintas a ARM.

También se podrían validar estas mejoras sobre otro tipo de sectores tecnológicos distintos al de las aplicaciones multimedia de forma que se ampliase el campo de estudio e investigación de ese tipo de dispositivos.

7. Bibliografía

[1] http://www.simplescalar.com/v4test.html Simplescalar v4.0 Tutorial

[2] http://www.eecs.umich.edu/~panalyzer/pdfs/ARM_doc.pdf The Simplescalar-ARM Power Modeling Project. ARM architecture documentation

[3] http://www.eecs.umich.edu/mibench/ MiBench Embedded Benchmark Suite

[4] A. Seznec. “A case for two-way skewed-associative caches”. In Proceedings of the 20th Annual International Symposium on Computer Architecture, May 1993

[5] P. Michaud, A. Seznec, and R. Uhlig, “Trading Conflict And Capacity Aliasing In Conditional Branch Predictors”. Computer Architecture, 1997. Conference Proceedings. The 24th Annual International Symposium on Volume , Issue , 2-4 Jun 1997

[6] A. Seznec, S. Felix, V. Krishnan, and Y. Sazeid`es. Design tradeoffs for the ev8 branch predictor. In Proceedings of the 29th Annual International Symposium on Computer Ahrchitecture, 2002.

[7] MCFARLING, S. 1993. *Combining branch predictors*. Tech. Rep. TN-36m, Digital Western Research Laboratory, June.

[8] A. N. Eden and T. Mudge, Dept. EECS, University of Michigan, Ann Arbor: The YAGS Branch Prediction Scheme In *Microarchitecture, 1998. MICRO-31. Proceedings. 31st Annual ACM/IEEE International Symposium*

[9] E. Sprangle, R. Chappell, M. Alsup, and Y. Patt, The Agree Predictor: A Mechanism for Reducing Negative Branch History Interference*. Proc. 24th Ann. Int. Symp. on Computer Architecture, May 1997.*

[10] LEE, C.-C., CHEN, C., AND MUDGE, T. 1997*. The bi-mode branch predictor.* In *Proceedings of the Thirtieth Annual International Symposium on Microarchitecture, 4–13.*

[11] Daniel A. Jiménez e Calvin Lin Department of Computer Sciences The University of Texas at Austin: Dynamic Branch Prediction with Perceptrons in *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*

[12] Chih-Cheng Cheng: “The Schemes and Performances of Dynamic Branch predictors”.

[13] P.-Y. Chang, E. Hao, T.-Y. Yeh, and Y.N. Patt. *Branch classification:a new mechanism for improving branch predictor performance*. In *Proceedings of the 27th Intemational Symposium on Microarchitecture, 1994*.

[14] http://www.uhu.es/raul.jimenez/EMPOTRADO/introduccion.pdf Definición y características de un Sistema Empotrado.

[15] Computer Architecture: A Quantitative Approach, by JL Hennessy, DA Patterson. 4th Edition (The Morgan Kaufmann Series in Computer Architecture and Design)

[16] ftp://ftp.ac.upc.edu/pub/reports/DAC/1996/UPC-DAC-1996-30.ps.Z
Recopilación de las técnicas más importantes en predicción dinámica de saltos aparecidas en la literatura en los últimos tiempos.

ANEXO I. CÓDIGO

```
struct bpred_dir_t {            /* direction predictor def */
  enum bpred_class class;       /* type of predictor */
  union {
    struct {
      unsigned int size;        /* number of entries in direct-mapped table */
      unsigned char *table;     /* prediction state table */
    } bimod;
    struct {
      int l1size;               /* level-1 size, number of history regs */
      int l2size;               /* level-2 size, number of pred states */
      int shift_width;          /* amount of history in level-1 shift regs */
      int xor;                          /* history xor address flag */
      int *shiftregs;                   /* level-1 history table */
      unsigned char *l2table;   /* level-2 prediction state table */
      int psize;                        /* number of registers in table of perceptrons */
      //      int pwidht;               /* amount of bits in perceptron table entry*/
      int **ptable;             /* perceptron table*/
      int pindex;               /* chosen perceptron pointer */
      int presult;              /* neural net outcome of the perceptron */
      unsigned int type;                /* level-2 predictor type  */
    } two;
    struct{
      int l1size;               /* level-1 size, number of history regs */
      int l2size;               /* level-2 size, number of pred states */
      int shift_width;          /* amount of history in level-1 shift regs */
      int xor;                          /* history xor address flag */
      int *shiftregs;                   /* level-1 history table */
      unsigned char *l2table;   /* choice PHT */
      struct {
  int sets;                     /* num BTB sets */
  int assoc;                    /* BTB associativity */
  struct bpred_btb_ent_t *cache_data;       /* cache data */
      } tcache;
      struct {
```

```
  int sets;                     /* num BTB sets */
  int assoc;                    /* BTB associativity */
  struct bpred_btb_ent_t *cache_data;       /* cache data */
      } ntcache;
      int lastUsed;
      int match;
      unsigned char PHTpred;
      int cacheIndex;
     }yagsScheme;
  } config;
};
```

Figura 7.1 Estructura de datos de la tabla de predicción

```
/* branch predictor def */
struct bpred_t {
  enum bpred_class class;     /* type of predictor */
  struct {
    struct bpred_dir_t *bimod;          /* first direction predictor */
    struct bpred_dir_t *twolev;         /* second direction predictor */
    struct bpred_dir_t *yags;    /* YAGS predictor */
    struct bpred_dir_t *meta;   /* meta predictor */
  } dirpred;
  struct {
    int sets;                   /* num BTB sets */
    int assoc;                  /* BTB associativity */
    struct bpred_btb_ent_t *btb_data; /* BTB addr-prediction table */
    //    int bias;
  } btb;
  struct {
    int size;                   /* return-address stack size */
    int tos;                    /* top-of-stack */
    struct bpred_btb_ent_t *stack; /* return-address stack */
  } retstack;
```

Figura 7.2 Estructura de datos del predictor de saltos

Mostramos ahora los fragmentos de código más significativo y más importantes a la hora de llevar a cabo la implementación de las técnicas antes descritas. En las figuras 7.1 y 7.2 vemos declaradas todas las estructuras de datos necesarias para alojar las tablas de predictores, para todos y cada uno de los predictores evaluados. Hay que destacar que para casi todos los que hemos implementado no han sido necesarias estructuras de datos nuevas, aunque sí alguna modificación. En cambio, tanto el predictor Perceptron, por su esquema novedoso y rompiendo la tónica con el resto, y el esquema Yags, sí precisaron de la declaración de estructuras de datos nuevas.

En la figura 7.3 mostramos las funciones hashing que hemos utilizado a la hora de indexar las tres partes en las que se divide la tabla de predictores del predictor Skewed. En dichas funciones se realizan diversos desplazamientos a nivel de bit. Mostramos también el fragmento de código donde se accede a cada parte del banco de predictores, con una función hashing distinta.

```
/* Definimos las funciones HASH para indexar las tres tablas del Predictor Skewed */


#define SKW_HASH(TUP, N)                                                \
 (((TUP & (1 << (N - 1))) ^ ((TUP & 1) << (N - 1))) | (TUP >> 1))


#define SKW_INV_HASH(TUP, N)                                            \
 (((TUP << 1) & ((1 << N) - 1)) | ((TUP >> (N - 1)) ^ ((TUP >> (N - 2)) & 1)))
```

Figura 7.3 Funciones hashing utilizadas

En la última figura se muestra la utilización de las funciones de hashing en la etapa de indexación de los bancos:

```
switch (ind_table)
          {
            case 0:
        {
          h = SKW_HASH(tup1, l2width);
          hinv = SKW_INV_HASH(tup2, l2width);
          p = &pred_dir->config.two.l2table[((h ^ hinv) ^ tup2)];
        }
         break;
       case 1:
         {
```

```
        h = SKW_HASH(tup1, l2width);
        hinv = SKW_INV_HASH(tup2, l2width);
        p = &pred_dir->config.two.l2table[pred_dir->config.two.l2size + ((h
^ hinv) ^ tup1)];
      }
      break;
    case 2:
      {
        h = SKW_HASH(tup2, l2width);
        hinv = SKW_INV_HASH(tup1, l2width);
        p = &pred_dir->config.two.l2table[(2 * pred_dir->config.two.l2size) +
((hinv ^ h) ^ tup2)];
      }
      break;
    default:
      break;
```

Figura 7.4 Indexación de la tabla de predictores en el Predictor Skewed