Recopilación de Técnicas de Predicción de Saltos

José González y Antonio González
Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya
c/Gran Capità s/n, 08071 Barcelona (Spain)

Email: {joseg,antonio}@ac.upc.es
Tel: +34 3 401 7187
Fax: +34 3 401 7055

Abstract

Los procesadores más utilizados en la actualidad son capaces de leer y ejecutar más de una instrucción por ciclo. Este hecho hace que los cambios en el flujo de ejecución de un programa (dependencias de control) supongan una degradación importante en el rendimiento debido a que el procesador, al encontrar un salto, ha de bloquear la búsqueda de instrucciones hasta que no sea ejecutado en su unidad funcional.

Una técnica utilizada para evitar estos bloqueos consiste en predecir el comportamiento de cada instrucción de salto (si será efectivo y su dirección destino) antes que sea resuelto. De esta manera el procesador no se bloquea y puede hacer la búsqueda de instrucciones en cada ciclo.

La predicción de saltos se puede realizar en tiempo de compilación (estática) o en tiempo de ejecución (dinámica).

En este trabajo se presentan las técnicas más importantes en predicción dinámica de saltos aparecidas en la literatura en los últimos tiempos. A su vez, se estudian las implementaciones que realizan sobre este tema los últimos procesadores aparecidos en el mercado.

INDICE

1 Introducción

En las máquinas actuales, las cuales son capaces de leer y ejecutar más de una instrucción por ciclo, las instrucciones de salto suponen un inconveniente para la obtención de un rendimiento alto. Estas instrucciones provocan *dependencias de control:* Cuando una instrucción de salto es decodificada, el procesador bloquea el fetch de instrucciones hasta que el salto no es resuelto, es decir, 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 de la cache de instrucciones (Icache), nos encontramos con que varios stages del pipeline están vacíos y que la ventana de instrucciones (donde están las instrucciones pendientes de ser ejcutadas) tiene pocas instrucciones. A este hecho se le llama introducir una burbuja en el pipeline. Esta burbuja provoca una bajada 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 métodos precisos que predigan la dirección de los saltos condicionales [13], así como anticipar lo antes posible el cálculo de la dirección destino.

La predicción de saltos 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 Icache y no introduciendo burbujas en el procesador.

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

- Las que realizan predicción estática.
- Las que realizan predicción dinámica.

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% [30]).

Existen también en la literatura una serie de técnicas, no basadas en la predicción, para reducir el coste de los saltos [9] [15]. Estas son por ejemplo: salto retardado, avance en el cálculo de las instrucciones de las que el salto depende, o la ejecución concurrente de los dos posibles caminos que se pueden coger al llegar a un salto condicional.

El objetivo de este trabajo es la recopilación de las técnicas dinámicas más conocidas, aunque se hará un pequeño resumen de algunas técnicas estáticas. La organización de lo que queda de trabajo será la siguiente: En el apartado 2 se repasan algunas técnicas de predicción estática. El apartado 3 presenta las técnicas más significativas de predicción dinámica. En el apartado 4 se ven como tienen implementada la predicción de salto algunos procesadores actuales. Finalmente en el apartado 5 se presentan las conclusiones.

2 Estrategias 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[21][14]:

- 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. Lee y Smith [14] hacen un estudio del comportamiento de los saltos según su código de operación para el IBM 370, en él se observa que para ciertos códigos de operación la probabilidad que sean tomados es bastante alta (del orden del 80%-90%)
- 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 Estrategias 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 des-

tino 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 [14][21]. 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. Ultimamente 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.

A continuación se presentará un repaso de las técnicas más importantes de predicción dinámica de salto: (1) Branch Target Buffer con información sobre la historia de cada salto, (2) predictores de dos niveles que usan la historia de cada salto o la interrelación de cada salto, (3) técnicas que predicen la siguiente línea de cache a leer, (4) predictores basados en el camino recorrido por el programa, y por último, (5) técnicas que combinan varios predictores

3.1 Branch Target Buffer

El Branch Target Buffer (BTB) es una pequeña memoria asociativa que guarda las direcciones de los últimos saltos ejecutados así como su destino. A su vez guarda información que permite predecir si el salto será tomado o no[14].

En la etapa de fetch se mira si la dirección de la instrucción está en el BTB. Si es así se miran los bits de predicción y se decide si el salto ha de ser tomado o no. Si el salto no es tomado o la dirección no está en el BTB en el siguiente ciclo se hace el fetch de la siguiente instrucción en orden. Si el salto es tomado en el siguiente ciclo se hace el fetch del nuevo camino de ejecución.

En cuanto a la política de entrada de saltos en el BTB, Perleberg y Smith [20] proponen en su trabajo la siguiente premisa: un salto que no tenga potencial para mejorar el rendimiento, no será introducido en el BTB. Para ello se pueden utilizar dos políticas: introducir un salto cuando es ejecutado por primera vez, o introducir un salto cuando es tomado. En su trabajo Perleberg concluye diciendo que para un BTB con un grado de asociatividad 4 el segundo método es más preciso (también es el que se ajusta más a su premisa). La premisa introducida por Perleberg para sacar saltos del BTB es: cuando sea necesario sacar un salto del BTB, descartar el que tenga menos potencial para mejorar el rendimiento. Bajo esta premisa el algoritmo propuesto sería un LRU (Least Recently Used), es decir sacar del BTB el salto que haga más tiempo que no ha sido referenciado. También podemos aplicar esta política únicamente sobre los saltos del BTB que tienen sus bits de predicción tal que el salto será predicho como no tomado.

Calder en su trabajo [2] realiza una optimización sobre lo anteriormente propuesto que sería introducir en el BTB sólo los saltos tomados. De esta manera, cuando un salto no está en el BTB de predice como no tomado. A su vez cuando un salto predicho como tomado realmente no es tomado, se invalida su entrada en el BTB.

Un aspecto importante a tener en cuenta es la colocación del BTB [1]. Este puede estar separado de la cache de instrucciones (Figura 1) o incorparado a la Icache (Figura 2). Observando estas figuras vemos que cada entrada del BTB contiene unos bits que almacenan la información de la historia del salto (Hb) y la dirección destino del salto (Target Address). Si el BTB está colocado junto con la Icache tiene como principales ventajas la facilidad en añadir el hardware adicional, y la no necesidad de duplicar tags, en cambio tiene como inconvenientes

que todos los saltos que pueda haber en una línea de cache comparten la misma información y la contención en el acceso a los tags, ya que se puede hacer este acceso en dos etapas diferentes del pipeline.

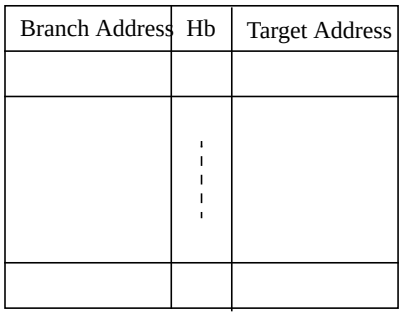


Figura 1.: BTB separada de la Icache

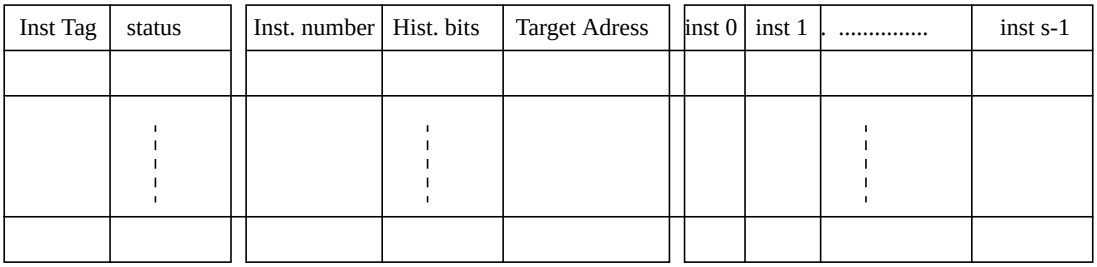


Figura 2: BTB incorporada a la Instruction Cache

Un punto clave en el diseño del BTB es el algoritmo de predicción, tanto por el número de bits empleados como por la forma de decidir la predicción y actualizar la información. El método más utilizado es el de un contador saturado de dos bits. Saturado quiere decir que cuando está en su valor máximo y lo incrementamos no pasa a cero, sino que se queda igual, a su vez cuando vale cero y lo decrementamos, su valor continua siendo cero.

El mecanismo de funcionamiento de este contador es el siguiente: cada vez que se tiene que hacer una predicción se mira el bit de más peso para decidir si el salto es tomado o no, y una vez que el salto es ejecutado se actualiza la información del contador de tal manera que si el salto ha sido tomado se incrementa y si el salto no ha sido tomado se decrementa. Esto significa que para valores altos del contador el salto será predicho como tomado, mientras que para valores bajos será predicho como no tomado.

Cuando el salto es ejecutado el procesador debe comprobar si la dirección previamente predicha es la correcta o no, es decir, ha de comprobar si el salto es tomado y también si la dirección destino que estaba en el BTB es la correcta. Normalmente la dirección será correcta, ya que mayoría de los saltos tienen siempre la misma dirección destino, pero hay un tipo de instrucción, el retorno de función, que cada vez puede saltar a una dirección diferente. Para esta instrucción hay propuesta en la literatura una estructura especial llamada *return stack*. Cuando

el procesador se encuentra una llamada a subrutina guarda en el *return stack* la dirección de retorno, y cuando el predictor detecta que la instrucción de salto es un return, en vez de coger la dirección destino del BTB la desempila del *return stack*.

Otro aspecto importante en el diseño de un BTB es su asociatividad. Un BTB de mapeo directo permite una velocidad de acceso mayor, aunque existe problema de contención: dos o más saltos pueden coincidir en una misma entrada. En este caso se pueden tomar dos decisiones: primero cuando un salto coincide con uno ya existente, quitamos el que había y reinicializamos el estado de los bits de predicción, la otra opción sería compartir los bits de predicción para todos los saltos que coinciden en una misma entrada del BTB. En un BTB con organización n-asociativa, tendremos que con menos entradas podemos conseguir un rendimiento semejante al de un BTB mayor de acceso directo. Para BTB completamente asociativos Perleberg y Smith [20] demuestran que a partir de 512 entradas el rendimiento obtenido crece escasamente.

El BTB es una buena estructura para predicción de saltos, ya que aunque no consigue porcentajes de acierto espectaculares, su coste en hardware es poco comparado con otros predictores.

3.2 Predictores basados en dos niveles de historia

Debido a la importancia de la predicción de saltos en el diseño de los computadores, ha resultado que el Branch Target Buffer basado en un nivel de predicción (contador saturado *up-down* que registra la historia de cada salto) no consigue un porcentaje de aciertos suficiente para las necesidades de los procesadores.

Diversos autores [26][19] han propuesto predictores más costosos a nivel de hardware que basan su predicción en dos niveles de historia: en un primer nivel recogen la historia de los *n* últimos saltos ejecutados o de las *n* últimas ejecuciones de un salto concreto (dependiendo de la implementación). Con esta información, en un segundo nivel, se indexa una tabla de patrones que en cada entrada guarda una máquina de estados (normalmente codificada en dos bits) con la que se decide si el salto es tomado o no.

Yeh y Patt [26], presentan una primera aproximación consistente en dos estructuras principales: Branch History Register Table (BHRT) y Branch History Pattern Table (BHPT) (Figura 3).

La BHRT es una tabla que contiene un registro de historia de m bits para cada salto. Este registro es de desplazamiento y almacena la historia de las últimas m ejecuciones de dicho salto. Por ejemplo, considerando T como salto tomado y N como salto no tomado, si un salto cualquiera ha tenido el siguiente comportamiento en sus últimas 5 ejecuciones, TNTTN, su registro de historia correspondiente será 10110. A la codificación binaria de este comportamiento se le llama patrón.

La BHPT es una tabla de 2^m entradas, indexada por el registro de historia. Esta tabla guarda la historia de las últimas ejecuciones de todos los saltos que tienen un mismo patrón, representado por el registro de historia.

Cuando llega un nuevo salto B_i, se obtiene su registro de historia de la BHRT, el contenido del cual es utilizado para indexar la Pattern Table, obteniendo S_c, que es la historia de la ejecución siguiente del salto cuando tenía el patrón dado por el registro de historia. Sobre estos bits es aplicada la función de predicción Z_c, la cual nos dice si el salto va a ser tomado o no.

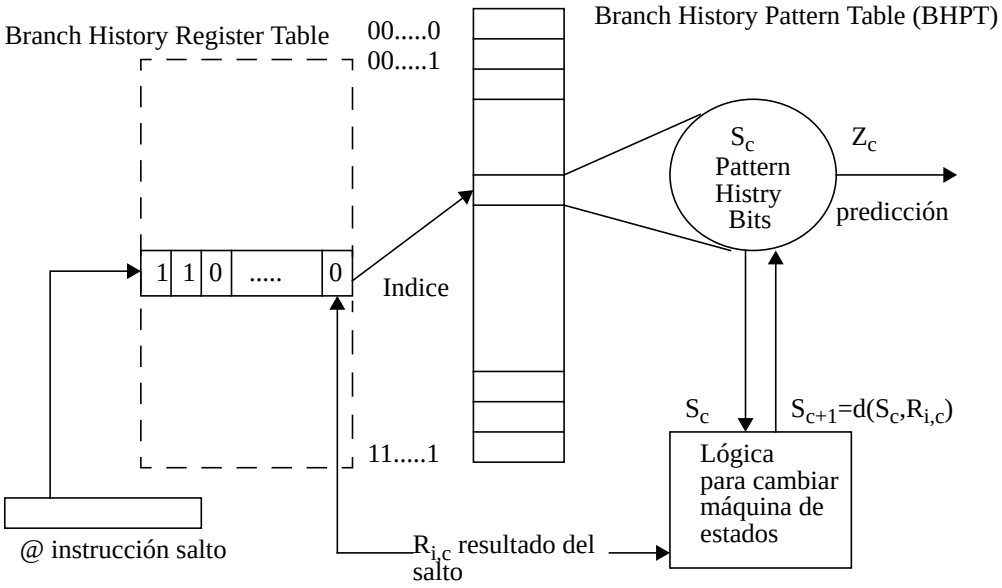


Figura 3: Estructura del predictor de dos niveles

Después que el salto es ejecutado, se actualiza el registro de historia del mismo colocando el resultado $R_{i,c}$ en el bit de menos peso del registro, desplazándose éste una posición hacia la izquierda, a su vez la entrada correspondiente en la Pattern Table es actualizada (S_{c+1}) mediante una función (d) que tiene como entradas la ejecución del salto ($R_{i,c}$) y el estado anterior (S_c). Dicha función puede ser un simple contador up-down de 2 bits que se incrementa cada vez que el salto es tomado y se decrementa cuando el salto no es tomado. Para realizar la predicción se puede coger el bit de más peso que indicará si el salto es efectivo o no.

Dos aspectos importantes a la hora de implementar este método es el grado de asociatividad de la tabla de registros de historia, así como la longitud en bits de cada registros. Yeh & Patt estudian dos modelos de asociatividad:

- Empleando una tabla hash. A ella se accede con unos cuantos bits de la dirección de la instrucción de salto. Esta tabla no contiene etiquetas, por lo que todos los saltos que tengan los mismos bits que sirven para indexar la tabla compartirán la información.
- Empleando una tabla n-asociativa.

Comparando modelos llegan a la conclusión que con una tabla de 512 entradas asociativa en grado cuatro se consiguen unos resultados óptimos sin hacer un gasto en hardware excesivo (consiguen un 97% de aciertos en simulaciones realizadas con los benchmarks SPEC 89 [26]).

En cuanto a la longitud de los registros es obvio que cuanto más grande sea el registro de historia más precisión se puede conseguir en las predicciones, Johnson [11] hizo un estudio sobre la influencia del tamaño del registro de historia en los esquemas con dos niveles de predicción, usando como cargas de prueba algunos programas de los SPEC 89 y una aplicación que realizaba la FFT. Su conclusión es que para conseguir una predicción del 97% es necesario un registro de historia global de 24 bits.

Después del primer trabajo de Yeh y Patt fueron apareciendo diferentes adaptaciones de su esquema, donde básicamente se variaba el número de registros de historia y tabla de patrones.

Pan *et al* [19] afirman que el camino tomado por un salto no sólo depende de su propia historia sino que también depende de la historia de los otros saltos, y por lo tanto proponen un método en el que usan la correlación entre saltos. Las diferencias con el método de Yeh y Patt consisten en que en el método basado en la correlación sólo encontramos un registro de historia y que utilizamos más de una tabla de patrones (por una sola que utiliza el esquema anterior).

El registro de historia, de longitud n, sirve para elegir una de las 2^n tablas de patrones, las cuales son indexadas con la dirección del salto. En la Figura 4 vemos un ejemplo donde se usa un registro de historia de 2 bits y cada entrada de la tabla de patrones tiene 2 bits de predicción (por ejemplo un contador up-down)

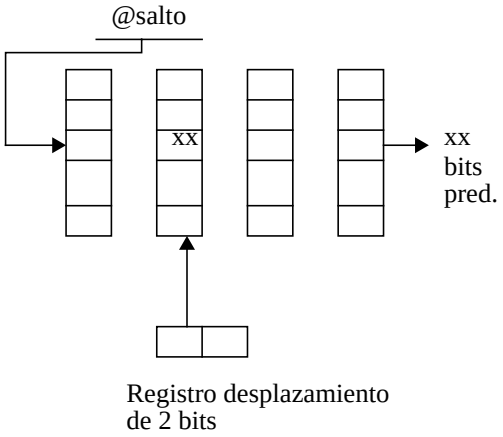


Figura 4: Esquema de correlación para un registro de historia de 2 bits

También aparecieron esquemas en que los saltos eran divididos en conjuntos y se asignaba un registro de historia a cada conjunto, utilizando los bits de menos peso de la dirección de la instrucción de salto para elegir el correspondiente registro de historia. También las tablas de patrones fueron variando de una global, a utilizar más de una.

Yeh y Patt [27][28] recopilaron los distintos métodos de predicción a dos niveles y crearon una nueva nomenclatura con la que se puede distinguir las diferentes implementaciones existentes de los predictores que usan dos niveles de predicción de salto. Esta nomenclatura se muesta en la Tabla 1.

Tabla 1: Nomenclatura de los esquemas con 2 niveles de predicción

Variación	Descripción	Coste Hard en bits
Gag (k,1)	Global Adaptive Branch Prediction using one global pattern history table.	$k + 2^k \times 2$
GAs(k,p)	Global Adaptive Branch Prediction using per-set pattern history tables.	$k + p \times 2^k \times 2$
GAp(k,b)	Global Adaptive Branch Prediction using per-address pattern history tables.	$k + b \times 2^k \times 2$
PAg(k,1)	Per-address Adaptive Branch Prediction using one global pattern history table.	$b \times k + 2^k \times 2$
PAs(k,p)	Per-address Adaptive Branch Prediction using per-set pattern history tables.	$b \times k + p \times 2^k \times 2$
PAp(k,b)	Per-address Adaptive Branch Prediction using per-address pattern history tables.	$b \times k + b \times 2^k \times 2$
SAg(k,s)	Per-Set Adaptive Branch Prediction using one global pattern history table.	$s \times k + s \times 2^k \times 2$
SAs(k,sxp)	Per-Set Adaptive Branch Prediction using per-set pattern history tables.	$s \times k + s \times p \times 2^k \times 2$
SAp(k,sxb)	Per-Set Adaptive Branch Prediction using per-address pattern history tables.	$s \times k + s \times b \times 2^k \times 2$

Las dos primeras letras describen la estructura de los registros de historia, es decir, un registro de historia global (GA), un registro de historia por cada salto (PA) o un registro de historia para cada conjunto de saltos (SA). La última letra describe la estructura de la tabla de patrones, que puede ser: una tabla de patrones global (g) una tabla de patrones para cada salto (p) o una tabla de patrones para cada conjunto de saltos(s).

El coste de estas estructuras en bits viene descrito en la Tabla 1 de una manera sencilla. Para cada implementación, k representa el número de bits que tienen los registros de historia, b representa el número de entradas en la tabla de registros de historia, p es el número de tablas de patrones y s es el número de conjunto de saltos.

En cuanto a los resultados, los esquemas de historia global (GA) tienen mejor rendimiento que los SA y PA en los programas de aritmética entera debido a que estos programas contienen muchas estructuras *if-then-else*, en las cuales la correlación entre saltos es alta. Por otra parte

en estos esquemas si el registro de historia es pequeño, los patrones entre diferentes saltos se interfieren y bajan el rendimiento, por lo tanto si se aplican esquemas de historia global es importante que los registros de historia sean lo más grande posible.

Los esquemas que tienen un registro de historia por dirección tienen mejor rendimiento en los programas de coma flotante, los cuales tienen un gran número de bucles, con un comportamiento periódico en los saltos. Por otra parte la historia de diferentes saltos tiende a influir menos, aunque es importante tener diferentes tablas de patrones, por ejemplo un esquema PAs (un registro de historia por salto y un conjunto de tablas de patrones).

En los programas enteros, los esquemas que tienen un registro de historia para cada conjunto de saltos tienen un rendimiento similar que los esquemas con un registro global de historia, mientras que en los programas de coma flotante obtiene unos resultados similares a los esquemas con un registro de historia por salto. Sin embargo, para que sean efectivos, el coste en hardware que necesitan es mayor.

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. 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 (estructura SA). 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*.

Young *et al* [29] estudian este hecho 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 la aplicaciones es el destructivo, y por lo tanto el rendimiento de la predicción baja considerablemente. Para evitarlo, hay que gastar más hardware en registros de historia y tener uno por cada salto.

Yeh y Patt [28] compararon los diferentes esquemas realizando simulaciones con los SPEC 89. Relacionando coste-efectividad el modelo que consigue mejores resultados con coste bajo es el PAs(6,16), fijando un coste máximo de 8Kbits consigue predicciones de hasta 96,2%. Aumentando el hardware destinado a predicción de salto a 128 Kbits, el modelo que más rendimiento da es el GAs(7,32), con registros de historia de 7 bits y 32 tablas de patrones, con el cual se consigue una predicción de hasta el 97.2%.

Como se ha visto, el uso de la historia global es menos eficiente que el uso de la historia local, aunque en cierto tipo de programas funciona bien. Eso hace pensar que el uso de un método que combinara los dos podría hacer mejorar el rendimiento. McFarling [16] propone dos esquemas donde combina la historia global con la dirección de cada salto

.

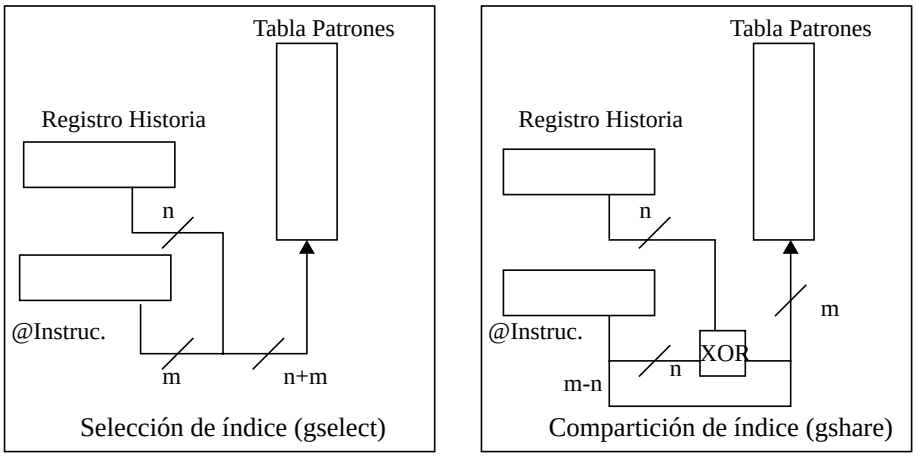


Figura 5: Esquemas de historia global combinados con la dirección del salto

Como podemos observar en la Figura 5, la tabla de patrones se indexa con una combinación del registro de historia y la dirección del salto. En el caso del *gselect,* se hace una simple concatenación de ambos, mientras que en el caso del *gshare* se realiza una operación XOR. Los resultados obtenidos con estos esquemas mejoran la precisión en la predicción de los esquemas globales, siendo el *gshare* un poco mejor que el *gselect.*

3.3 Predicción de la siguiente línea de cache

Los predictores de la siguiente línea de cache (Next cache Line and Set) han aparecido en la literatura[3][12] como alternativa al Branch Target Buffer. Un predictor NLS nos da un puntero a la cache de instrucciones indicando donde se encuentra la instrucción destino.

Johnson [12] propone un método en el que en cada línea de cache se añade información sobre la línea de cache que viene a continuación si el salto es tomado, la instrucción destino dentro de la nueva línea de cache y también qué posición ocupa el salto dentro de la línea actual, de tal manera que las instrucciones situadas a continuación no deben ser ejecutadas.

Calder y Grunwald [3] proponen un método más complejo, donde la línea destino también depende del tipo de salto (si es un retorno de subrutina se lee del *return stack*), y donde utilizan el método de dos niveles explicado anteriormente para hacer la predicción (Figura 6)

Cuando el procesador detecta una instrucción de salto en la etapa de fetch, con los bits de menos peso de la dirección de la instrucción se accede a la entrada correspondiente de la NLS. Allí se mira el campo *tipo de instrucción de salto* para poder escoger entre todas las posibles direcciones destino. Si es un retorno de subrutina, se leerá del return stack, si es un salto incondicional o condicional se leerá de la entrada correspondiente de la NLS. En el caso de los saltos condicionales, también se accederá al predictor de dos niveles para saber si el salto será

tomado o no. Si no fuera tomado, la siguiente instrucción a ejecutar sería la actual más el número de instrucciones que se leen en la etapa de fetch, tal y como viene indicado en la Figura 6

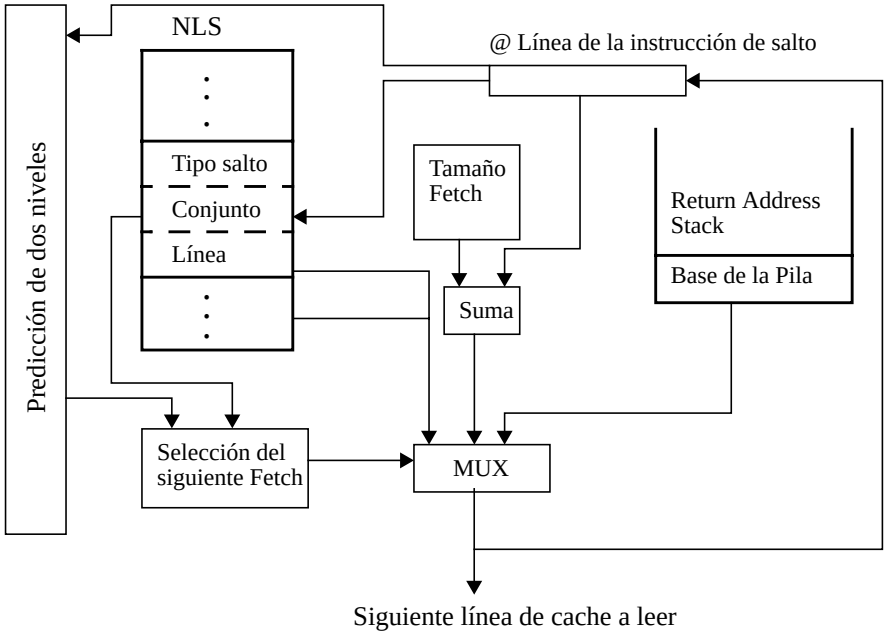


Figura 6: Esquema de la arquitectura NLS-table

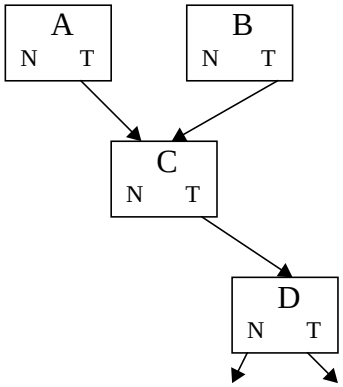
Las entradas del NLS son actualizadas después que las instrucciones son decodificadas y el tipo de salto y el destino es resuelto. Sólo se modifican los campos conjunto y línea en los saltos tomados, mientras que el campo tipo de salto es escrito para todos los saltos.

En su trabajo Calder y Grunwald distinguen dos tipos de NLS: NLS-table, que es la descrita anteriormente y NLS-cache, introducida por Johnson[12]. Esta segunda incorpora los bits de predicción a la cache, pudiendo predecir dos saltos por cada línea de cache (para ellos una línea son ocho instrucciones). La NLS-table tiene tres ventajas y un inconveniente sobre la otra estructura. Primero es que en la NLS-cache podemos encontrar líneas sin saltos y por lo tanto desperdiciar predictores, en cambio la NLS-table indexa la tabla de predicción desacoplada con los bits de menos peso de la instrucción de salto. Segundo es que cuando hay un reemplazo de la línea de cache: en el caso de la NLS-cache se descarta toda la información sobre las predicciones. Tercero, el tamaño de la NLS-cache es mayor, y por lo tanto su diseño es más costoso para caches grandes. La desventaja es que en la NLS-table podemos encontrar diversos saltos que colisionen sobre la misma entrada del predictor, y por lo tanto haya interferencia entre saltos.

Comparando resultados, Calder y Grunwald llegan a la conclusión que entre los dos diseños de NLS (table y cache) es mejor el NLS-table, teniendo en cuenta además que en la NLS-cache aumenta el número de predictores linealmente respecto al tamaño de cache, lo que hace al modelo poco escalable. Comparando BTB y NLS-table, sus estudios demuestran que este segundo modelo mejora el rendimiento del primero para estructuras que tienen el mismo coste en hardware (comparan un BTB de 128 entradas y una NLS con una tabla de 1024 entradas).

3.4 Predictores basados en el camino recorrido

Este esquema presenta una predicción en dos niveles, donde en el primer nivel no se guarda la historia de los último saltos ejecutados, sino el camino por donde ha ido pasando la ejecución. Consideremos el diagrama de bloques básicos[30]:



Camino ACD
Patrón “TT”
Probabilidad tomado salida D = 33%

Camino BCD
Patrón “TT”
Probabilidad tomado salida D = 89%

En este ejemplo la salida del bloque básico D no depende tanto de la historia de los últimos saltos ejecutado, como del camino recorrido. Vemos que para un mismo patrón (TT) la salida de D puede ser un salto tomado con poca probabilidad (33%), si venimos por los bloques AC, o un salto tomado con mucha probabilidad (89%) si venimos por los bloques BC. Por lo tanto utilizando registros de historia, sobre el salto de salida del bloque básico D frecuentemente se hará una predicción errónea. Sin embargo si guardamos información del camino recorrido, es decir, de los bloques básicos ejecutados anteriormente, tendremos una predicción más precisa de la ejecución del salto de salida del bloque básico D.

En la literatura encontramos diversos esquemas que hacen predicción basándose en el camino del programa, algunos hacen una predicción en tiempo de compilación[30], otros realizan una predicción dinámica con un esquema hardware similar al de los predictores basados en la historia de los saltos.

Nair [18], propone un esquema (Figura 7) similar al que presentan Patt *et al*. La diferencia radica en que en el esquema de Nair el registro de historia (*history register*), que guarda la ejecución de los últimos *p* saltos, es reemplazado por un registro que guarda el camino recorrido (*g-bit path history register*), es decir los *p* últimos bloques básicos recorridos. El funcionamiento de este registro es similar al esquema de Yale y Patt, pero aquí el registro de desplazamiento (compuesto por varias filas de bits), es actualizado con unos cuantos bits de la dirección destino del salto, los suficientes para indicar el bloque básico al cual pertenece. Con este registro se elige una de las tablas de patrones de donde se extraerá la información (en modo de contador saturado up-down) que nos permitirá realizar la predicción.

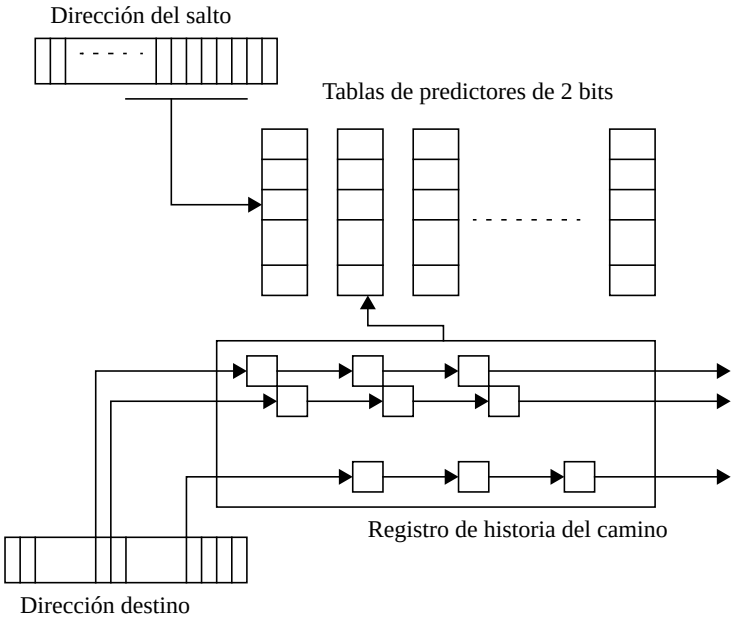


Figura 7: Esquema de la organización basada en la historia del camino recorrido

Podemos observar que en este esquema nos volvemos a encontrar con el problema del aliasing, ya que al no poder utilizar todos los bits que quisiéramos para guardar el camino, por problemas de hardware, nos podemos encontrar casos en los que no podremos distinguir dos caminos distintos que nos conducen a un salto.

Finalmente comparando los dos esquemas, el basado en la historia de los saltos y el basado en el camino recorrido, Nair llega la conclusión que tienen un rendimiento similar, siendo más útil el esquema basado en el camino recorrido en aplicaciones comerciales (con un comportamiento irregular de los saltos) y el esquema basado en el patrón de historia para aplicaciones científicas.

3.5 Predictores híbridos

Los diferentes predictores vistos hasta ahora tienen sus ventajas y sus inconvenientes, lo cual hace pensar que si se pudiera combinar su uso podríamos sacar partido de sus características y conseguir mayor precisión en la predicción. Esta es la filosofía de los predictores híbridos, que fueron propuestos por primera vez por McFarling [16].

Su esquema consiste en un conjunto de predictores y un mecanismo de selección entre ellos. Para cada salto todos los predictores realizan su predicción, y el selector elige el que hasta entonces haya tenido mejores resultados. Su propuesta es utilizar dos predictores y una tabla de contadores saturados up-down de dos bits, que sería indexada por la dirección de la instrucción de salto (Figura 8). Cada contador sería el encargado de seleccionar el predictor adecuado para cada salto, basándose en los aciertos o fallos que han tenido los predictores hasta entonces.

Si en la última predicción del salto P1 y P2 fallaron el contador no se cambia, si en cambio P1 falló y P2 acertó el contador es decrementado. Si P1 acertó y P2 falló el contador se incrementa, y por último si los dos aciertan el contador no se toca. Con el bit de más peso del contador se elige P1 (si vale 1) o P2 (si vale 0), ya que si el contador tiene un número alto significa que P1 ha acertado más veces que P2, y si tiene un número bajo significa que P2 ha acertado más veces que P1.

El siguiente paso sería escoger qué predictores se utilizarán. McFarling concluye su trabajo diciendo que el mejor sistema sería utilizar un predictor que usa historia local (por ejemplo un BTB con un contador de dos bits para realizar la predicción), junto con otro basado en la historia global, como el gshare. Con este sistema llega a conseguir hasta un 98.1% de aciertos en la predicción realizando las simulaciones sobre los 10 primeros millones de instrucciones del conjunto de programas SPEC89.

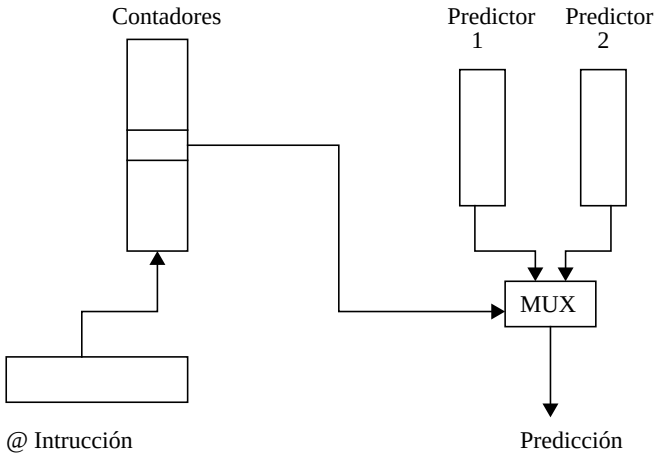


Figura 8: Esquema de predictor híbrido propuesto por McFarling

Chang et al [4] hacen un estudio más estricto tanto de los predictores a utilizar como del método de selección de los mismos. Los predictores que utiliza en su trabajo son:

- Un predictor estático basado en profiling.
- El predictor de un nivel basado en una lista de contadores de dos bits.
- El esquema introducido por Yeh y Patt PAs(m,n), que tiene 1K registros de historia de m bits y n tablas de patrones.
- El predictor creado por McFarling basado en la historia global (*gshare*).

Haciendo combinaciones de estos cuatro predictores llega a la conclusión que la mejor predicción se consigue utilizando los predictores *gshare* y PAs.

Respecto al mecanismo de selección de los predictores, proponen una técnica llamada algoritmo de selección del predictor del salto de dos niveles (parecido al creado por Yeh y Patt para la predicción de saltos). Su esquema se puede ver en la Figura 9.

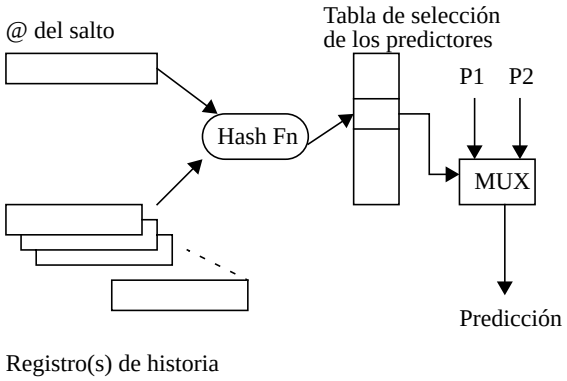


Figura 9: Estructura del selector de dos niveles

En el primer nivel, a partir de la dirección del salto y de su registro de historia, se elige una de las entradas del segundo nivel, que es una tabla de contadores con un comportamiento como el anteriormente explicado, que seleccionan la estructura de predicción que se quiere usar para ese salto en función de la que mejor comportamiento está teniendo hasta entonces. Si alguno de los predictores usados tiene una estructura de dos niveles de predicción, por ejemplo PAs, se puede compartir la tabla de registros de historia y la función de hash entre el predictor de dos niveles y la estructura de selección del predictor. Para la función de hash, se puede utilizar un XOR, parecido al gsahre de McFarling.

Evers, Chang y Patt[7], presentan un nuevo predictor híbrido que utiliza más de dos predictores individuales, realizando además un estudio de la influencia de los cambios de contexto en la predicción de salto, y cómo un predictor híbrido puede paliar el descenso del rendimiento de los predictores debido a dichos cambios.

Los cambios de contexto son un elemento importante a la hora de estudiar las diversa técnicas de predicción de salto, ya que provocan una bajada en el porcentaje de aciertos debido a que en cada cambio de contexto se puede hacer un vaciado de las diversas estructuras usadas para hacer la predicción[18][20], por lo tanto se pierde toda la historia guardada. Incluso no haciendo el vaciado de las tablas, la interferencia entre los saltos de distintos procesos hacen bajar el rendimiento de los predictores. Gloy et al[8], hacen un estudio sobre el efecto que tiene en la predicción de saltos la interacción entre flujos de ejecución de usuario y flujos de sistema, llegando a la conclusión que esta interacción hace bajar el rendimiento hasta un 10%.

En su diseño, Evers *et al* proponen añadir a cada entrada del BTB *N* contadores de 2 bits (1 por cada estructura de predicción usada). Para cada entrada nueva en el BTB, se inicializa su contador a 3. Cuando llega un salto todos los predictores realizan la predicción, escogiendo la predicción de aquel que tenga el valor tres en su contador. Si hubiera más de uno, se utilizará un codificador con prioridad para elegir el predictor.

Cuando el salto es resuelto se actualizan todos los contadores, si uno de los predictores que tenía valor tres en su contador ha acertado la predicción, decrementamos el valor de los otros contadores que hayan fallado la predicción. En caso contrario se incrementará el valor de todos los contadores, de esta manera siempre habrá algún contador que tenga el valor tres.

A la hora de escoger los predictores a usar, han tenido en cuenta la pérdida de rendimiento debida a los cambios de contexto. Por ello han combinado el uso de predictores capaces de almacenar mucha historia, pero ineficaces cuando se hace el cambio de contexto debido a que se pierde toda la historia al vaciarse las estructuras, con predictores estáticos y dinámicos con poca historia, eficaces en los momentos del cambio de contexto.

Los predictores estáticos usados son:

- Siempre tomado: Se predicen todos los saltos como tomados.
- *Loop*: Basado en el profiling: La predicción de la salida de los bucles se basa en el número de iteraciones que ha realizado el bucle en las ejecuciones anteriores. Una tabla de 2K de longitud de contadores de 8 bits es usada para mantener los contadores de las iteraciones.

Los predictores dinámicos usados son:

- 2bC: Una tabla de 2k entradas con contadores de 2 bits.
- GAs(m,n): Predictor de dos niveles con un registro global de *m* bits y *n* tablas de patrones.
- *gshare*: Modificación hecha por McFarling de los predictores de dos niveles que usan una historia global.
- *pshare*: Modificación del predictor de dos niveles que usa un registro de historia por dirección. Tiene una tabla de 2K registros de historia y como en el *gshare* se realiza una XOR del registro de historia con la dirección del salto para indexar la tabla de patrones.

Evers *et al* tomaron resultados a partir de la simulación de los SPECint92. Sin tener en cuenta los cambios de contexto el predictor multihíbrido mejora el rendimiento del híbrido PAs/gshare, obteniendo unas predicciones del 97.13% de media, frente al 96.63% de media del híbrido más simple. Teniendo en cuenta los cambios de contexto la diferencia se hace un poco mayor, ya que el multihíbrido obtiene un 96.22%, mientras que la combinación PAs/gshare obtiene el 95.22%, utilizando ambos la misma cantidad de hardware.

Una de las principales ventajas del predictor multihíbrido es la combinación de predictores que tienen comportamiento distinto según la cantidad de historia almacenada. Esto es importante cuando se tiene en cuenta los cambios de contexto, ya que el vaciado de las tablas hace perder toda la historia almacenada, y es donde los predictores que no están basados en una gran cantidad de historia obtienen un mejor rendimiento.

4 Predicción de saltos en los procesadores actuales

4.1 MIPS R8000 (TFP)

El TFP implementa un sistema de predicción de salto similar al NLS de Calder y Grunwald. Dispone de una Branch prediction cache(Bcache)[24], que es accedida en paralelo junto con la Icache en la etapa de fetch.

La Bcache tiene 1K entradas y es de mapeo directo. La salida de esta cache contiene un bit de predicción, la línea de cache de la instrucción destino y la posición que ocupa la instrucción dentro de la línea.

La arquitectura MIPS introduce un *delay slot* después de cada instrucción de salto, por lo tanto sólo puede haber dos saltos en una línea de cache. Teniendo en cuenta que el compilador puede hacer optimizaciones para separar los saltos entre sí, las posibilidades de tener dos saltos en una línea de cache es muy baja.

Cuando el salto es resuelto se compara la dirección destino calculada con la de la instrucción que sigue al *delay slot* (la instrucción destino predicha). Si ha habido un fallo de predicción el procesador tarda tres ciclos en solucionarlo: vaciado de las etapas de fetch, decodificación y cálculo de direcciones, a la vez que se hace el fetch del camino correcto.

Una característica importante de este esquema es que su coste hardware no es muy alto ya que en la Bcache se guardan sólo 10 bits que indican la línea destino, no toda la dirección virtual. Además para incrementar la eficiencia se realiza la predicción en la etapa de fetch.

4.2 MIPS R10000

El procesador R10000 implementa una tabla de 512 contadores de 2 bits. Esta tabla está mapeada directamente por los bits 11:3 de la dirección de la instrucción de salto y representan la historia reciente del salto. Utilizando contadores saturados up-down se consigue una precisión del 87% de aciertos[25].

Cuando el procesador, en la etapa de decodificación, predice un salto como tomado, descarta todas las instrucciones que haya leído en la etapa de fetch que vengan después del salto. El R10000 carga la dirección instrucción destino en el *Program Counter* y hace el fetch de las nuevas instrucciones después de un ciclo de retardo. Esto introduce una burbuja en el pipeline del procesador, durante la cual no decodifica ninguna instrucción.

Un aspecto importante de este procesador es la manera en la que guarda el estado cuando hace la predicción de una salto. El R10000 guarda en una estructura interna la dirección alternativa del salto, copias de los bancos de registros lógicos y físicos, así como una serie de bits de control. El objetivo de esta estructura es que en el caso de un fallo de predicción se pueda deshacer todo lo ejecutado en un ciclo.

4.3 HP - PA8000

El procesador PA8000 permite el uso de predicción estática o dinámica[10]. Con un bit extra de control en el TLB se selecciona el tipo de predicción que se utilizará.

En el modo de predicción estática, la unidad de fetch sigue la siguiente política: Para las instrucciones de comparación y salto, en el código de operación hay un campo que indica a la unidad de fetch si ha de saltar o no.

En cuanto a la predicción dinámica, el procesador PA8000 utiliza dos estructuras:

- Branch History Table: tabla de 256 entradas, que tiene en cada una de ellas un registro de desplazamiento de 3 bits que indica la historia de las 3 últimas ejecuciones de ese salto. Si la mayoría de las veces ha sido tomado, se predirá el salto como tomado. La tabla es actualizada cuando los saltos son retirados del pipeline para evitar la polución de saltos ejecutados especulativamente.
- Branch Target Address Cache: Es una tabla completamente asociativa que guarda la dirección destino de los 32 últimos saltos ejecutados. Se introduce un salto cada vez que es tomado y se aplica una política de *round-robin* para reemplazar saltos.

Cuando en la etapa de fetch una instrucción está en la BHT y es predicha como tomada, se busca si su dirección destino está en la BTAC, y si es así en el siguiente ciclo se podrá hacer el fetch del camino predicho.

4.4 ALPHA 21164

El procesador 21164 implementa una estructura de predicción de saltos dinámica[6]. Para ello mantiene una estructura de 2048 contadores *up-down* saturados, indexados con los bits de menos peso del *Program Counter.* Los contadores se incrementan cuando el salto es tomado y se decrementan cuando el salto no es tomado. Un salto es predicho como tomado si el bit de más peso está a 1.

Para el cálculo de las direcciones destino dispone de 4 sumadores (al realizar el fetch de 4 instrucciones cualquiera de ellas puede ser un salto) con desplazamiento que producen el destino teórico del salto en paralelo con el cálculo de la predicción.

Durante el segundo ciclo del pipeline, el procesador examina el bloque de instrucciones que le llega de la Icache o del *refill buffer.* Chequea las instrucciones de salto, su predicción y calcula la dirección destino para que se pueda leer en el siguiente ciclo, creando por lo tanto un ciclo de retardo (burbuja) en el pipeline.

4.5 AMD-K5

El procesador AMD-K5 implementa un sistema de predicción insertado en la Icache [5], similar al explicado por Johnson en su trabajo[12].

En cada línea de la Icache hay información sobre si el salto es tomado o no (si hay más de un salto en la línea éstos comparten la información), la siguiente línea a leer y la instrucción dentro de la línea a leer.

La predicción puede fallar por dos motivos: La dirección predicha sea diferente de la después calculada, o que el bloque destino no esté en la cache.

Con esta implementación el hardware gastado es el 25% mayor del gastado por un BTB de 256 entradas asociativo en grado 4.

4.6 SUN ULTRASPARC I

El procesador UltraSparc tiene una estructura de predicción de saltos incorporada a la cache de instrucciones[23], similar a la del procesador AMD-K5.

Dentro de la Icache incorpora un campo que indica la línea de cache destino, y dos bits con los que implementa una máquina con 4 estados: no-tomado, probablemente no-tomado, probablemente tomado, y fuertemente tomado. Cada vez que un salto es tomado se incrementa el valor de los bits y cuando no es tomado se decrementa.

La inicialización de la máquina de estados la realiza de dos maneras: Para alguna instrucciones de salto deja que el compilador, a través de un bit en la instrucción, inicialice el contador de dos bits a probablemente-tomado o probablemente no-tomado. Para el resto de instrucciones lo inicializa como probablemente no-tomado.

5 Conclusiones

En este trabajo se han visto las técnicas más importantes en predicción dinámica de saltos encontradas en la literatura en los últimos años, así como la implementación de estas técnicas en los procesadores actuales.

Un primer punto clave es comparar las técnicas dinámicas con las estáticas. En la literatura van apareciendo técnicas estáticas similares a las dinámicas, con la única diferencia que la máquina de estados ya viene determinada en tiempo de compilación[8]. Esto hace que el coste hardware sea menor, pero también hace que no siempre se consiga un buen porcentaje de predicción. Estudios recientes [17] demuestran que el comportamiento de los saltos en un programa varían sustancialmente dependiendo de la carga de entrada. Por lo tanto es lógico pensar que para este tipo de programas los métodos estáticos (que usan profiling no pueden conseguir resultados tan buenos como los dinámicos.

Otro punto importante a analizar son los benchmarks utilizados por los diseñadores para analizar las implementaciones presentadas. Sechrest *et al* [22] demuestran en su trabajo que el utilizar sólo los SPEC como carga de prueba (cosa que hacen la mayoría de los autores) no es una buena opción ya que tienen un comportamiento bastante similar en cuanto a saltos: pocos saltos que se ejecutan muchas veces. Por ello en este trabajo no se han incluido tablas comparatorias, ya que cada autor a la hora de sacar estadísticas utilizaba cargas de prueba diferentes.

En cuanto a las técnicas implementadas por los procesadores, nos encontramos con que distan mucho de parecerse a las propuestas en los últimos años (excepto los que usan la técnica de predecir la siguiente línea de cache). No existe ningún procesador que intente implementar técnicas basadas en dos niveles de historia y básicamente lo que hacen es añadir el predictor a la cache de instrucciones o crear un Branch Target Buffer dividido en dos estructuras: (1)Branch

history Table que utilizando contadores de dos bits realiza una predicción sobre si el salto va a ser tomado o no. (2) Branch Target Address Cache: donde se guarda la dirección destino del salto.

Actualmente los diseñadores de procesadores, aun sabiendo que es un factor que afecta al rendimiento del procesador, no gastan mucho hardware para implementar técnicas de predicción de saltos, y como con poco hardware todas las técnicas obtienen resultados parecidos, se implementan las más sencillas.

6 Referencias

[1] B. K. Bray and M.J. Flynn: Strategies for Branch Target Buffers. In *Proceedings of the 24th Annual International Symposium on Microarchitecture*, pp 42-50, 1991.

[2] B. Calder and D. Grunwald: Fast & Accurate Instruction Fetch and Branch Prediction. *In Proceedings of 21th Annual International Symposium on Computer Architecture*, pp 2-11, 1994.

[3] B. Calder and D. Grunwald: Next Cache Line and Set Prediction. In *Proceedings of 22th Annual International Symposium on Computer Architecture*, pp 287-295, 1995.

[4] P. Chang, E. Hao and Y. Patt: Alternative Implementations of Hybrid Branch Predictors. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pp 252-257, 1995

[5] D. Christie: Developing the AMD-K5 Architecture. In *IEEE Micro* Volume 16, Number 2, pp 16-26, 1996.

[6] J.H. Edmonson, P.Rubinfeld, R.Preston. V.Rajagoplan: Superscalar Instruction Execution in the 21164 Alpha Microprocessor. In *IEEE Micro* Volume 15, Number 2, pp 33-43, 1995.

[7] M.Evers, P. Chang, Y. Patt: Using Hybrid Branch Predictors to Improve Branch Prediction Accuracy in the Presence of Context Switches. In *Proceedings of 29th Annual International Symposium on Computer Architecture*, pp 3-11, 1996.

[8] N. Gloy, C.Young, J. Bradley and M.D. Smith: An Analysis of Dynamic Branch Prediction Schemes on System Workloads. In *Proceedings of 29th Annual International Symposium on Computer Architecture*, pp 12-20, 1996.

[9] A. González: A Survey of Branch Techniques in Pipelined Processors. In *Euromicro Journal*, pp 243-257, 1993.

[10] D. Hunt: Advanced Performance Features of the 64-bit PA-8000. In *Proceedings of CompCon 95*, pp 123-128, 1995.

[11] J. D. Johnson: Branch Prediction Using Large Self History. *Dept. of Elelctrical Engineering and Computer Science Tech. Report CSL-TR-92-553*, Standford University, Dec 1992.

[12] M. Johnson: Superscalar Microprocessor Design: Publicado por Prentice-Hall Inc, 1991.

[13] M.S. Lam and R.P. Wilson: Limits on Control Flow Parallelism. In *Proceedings of 25th Annual International Symposium on Computer Architecture*, pp 46-57, 1992.

[14] J. Lee and A.J. Smith: Branch Prediction Strategies and Branch Target Buffer Design. In *Computer*, Volume 17, Number 1, pp 6-22, 1984

[15] D.L. Lilja: Reducing the branch penlty in pipelined processors, *IEEE Computer 21 (7)*, pp 47-55, Julio 1988.

[16] S.MacFarling, Combining Branch Predictors, *WRL Techincal Note TN-36, Digital Equipment Corp.*, Jun. 1993.

[17] T. N. Mudge, I. K. Chen and J. T. Coffey: Limits to Branch Prediction. *Computer Science and Engineering Div. Tech.Report CSE-TR-282-96*,Univ. of Michigan,Ann Arbor, 1996.

[18] Ravi Nair: Dynamic Path-Based Branch Correlation. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pp 15-23, 1995.

[19] S. Pan , K. So and J. Rahmeh: Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation. In *Proceedings of the 5th Annual International Conference on Architectural Support for Programming Languages and Operating Systems*, pp 76-84, 1992.

[20] C. H. Perleberg and A.J. Smith: Branch Target Buffer Design and Optmization. In *IEEE Transactions on Computers*, Volume 42, Number 4, pp 396-412, April 1993.

[21] J.E. Smith: A Study of Branch Prediction Strategies: In *Proceedings of the 8th Annual International Symposium on Computer Architecture*, pp 135-148, 1981.

[22] S. Sechrest, C. Lee, and T. Mudge: Correlation and Aliasing in Dynamic Branch Predictors. In *Proceedings of 29th Annual International Symposium on Computer Architecture*, pp 22-32, 1996

[23] M. Tremblay and J.M. O'Connor: UltraSparc I: A Four-Issue Processor Supporting Multimedia. In *IEEE Micro* Volume 16 Number 2, pp 42-49, 1996.

[24] P. Hsu: Designing the TFP Microprocessor: In *IEEE Micro* Volume 14 Number 2, pp 23-33, 1994.

[25] K. Yeager: The MIPS R10000 Superscalar Microprocessor. In *IEEE Micro* Volume 16 Number 2, pp 28-40, 1996.

[26] T.Yeh and Y. N. Patt: Two Level Adaptative Branch Prediction. In *Proceedings of the 24th Annual International Symposium on Microarchitecture*, pp 51-61, 1991.

[27] T. Yeh and Y. N. Patt: Alternative Implementations of Two-Level Adaptive Branch Prediction. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pp 124-134, 1992.

[28] T. Yeh and Y. N. Pat: A Comparison of Dynamic Branch Predictors that use Two Levels of Branch History. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pp 257-266, 1993.

[29] C. Young, N. Gloy and M. D. Smith: A Comparative Analysis of Schemes for Correlated Branch Prediction. In *Proceedings of 22th Annual International Symposium on Computer Architecture*, pp 276-286, 1995.

[30] C. Young and M.D. Smith: Improving the Accuracy of Static Branch Prediction Using Branch Correlation. In *Proceedings of the 6th Annual International Conference on Architectural Support for Programming Languages and Operating Systems*, pp 232-241, 1994.