ESCOLA TÈCNICA SUPERIOR
D'ENGINYERIA
Universitat Rovira i Virgili

Práctica 2: Predictores de salto

Alumnos: Alain Martínez

Antoni Llop

Carrera: Ingeniería Informática

Profesor: Carles Aliagas

Fecha: Noviembre 2025

1. INTRODUCCIÓN

Predictores de salto

Los predictores de salto son un componente esencial para mejorar el rendimiento. Su función es la de anticipar un camino, es decir, cada vez que el flujo puede seguir ramas distintas, se encarga de decidir de forma especulativa cuál se va a tomar. Esto se hace ya que, en la etapa de frontend, es donde se recogen las instrucciones para alimentar el resto de etapas del procesador. Pero no es ahí donde se resuelven, es en el Execution Core. Por ende, para elegir el camino correcto, habría que detener el pipeline y esperar a conocer el resultado final.

El predictor trata de “adivinar” qué camino se va a elegir y rellenar el pipeline con sus instrucciones. En el caso de que no se acierte, hay que vaciar el pipeline completo y cargarlo con las instrucciones pertinentes. En el caso de que sí se acierte, se sigue normal, habiendo ganado tiempo.

Estos predictores se añaden ya que es más eficiente equivocarse de vez en cuando y tener que hacer un flush completo (vaciar el pipeline) que no hacer nada.

Un buen predictor es aquel que tiene, evidentemente, una tasa de acierto elevada. Además, reduce los riesgos de control y aumenta el rendimiento (sube el IPC). Algunas de las herramientas que utiliza para llevar a cabo esto son:

- BHR (Branch History Register) = mantiene el historial reciente de saltos. Guarda por cada instrucción de salto si las últimas ejecuciones han sido *taken* / *not taken*. Puede ser global (GBHR) o por dirección (PaBHR).

- PHT (Pattern History Table) = se obtienen patrones binarios, que representan si los últimos saltos han sido tomados o no tomados y para cada uno de ellos hay un contador saturado de 2 bits que determina si se toma o no el próximo salto.

- BTB (Branch Target Buffer) = almacena direcciones destino de instrucciones de salto que se han tomado ya. Es una combinación de BHR con PHT, y dicha combinación nos permite saber si hacemos taken / not taken, y además, si hacemos taken, a dónde saltamos.

- RAS (Return Address Stack) = guarda direcciones de retorno de llamadas a funciones.

A continuación, se hará un análisis de los resultados obtenidos para diferentes predictores de salto y explicaremos las ventajas y desventajas de cada uno.

2. PREDICTORES

PREDICTORES ESTÁTICOS

Introducción

Los **predictores estáticos** son los más simples de todos. Básicamente, no aprenden nada del comportamiento pasado del programa: siempre predicen lo mismo. Tienen una forma de actuar fija. Pueden parecer muy básicos (y lo son), pero sirven como punto de partida para comparar con los demás predictores más avanzados.

Tenemos dos variantes principales:

- **Not Taken**, que asume que ningún salto se tomará, es decir, que el programa seguirá ejecutando las instrucciones de forma secuencial.

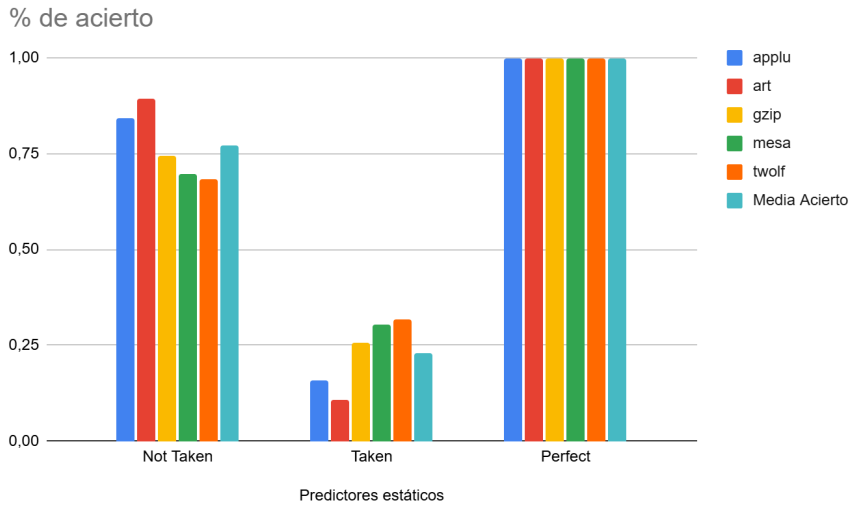
- **Taken**, que hace justo lo contrario: asume que todos los saltos se tomarán.

También incluimos el **Perfect**, que no es un predictor real, sino un caso ideal en el que el procesador “adivina” siempre correctamente. Sirve para saber el límite máximo de rendimiento que podríamos alcanzar si el predictor fuera perfecto.

Aunque estos métodos parezcan menos efectivos, en algunos casos pueden funcionar sorprendentemente bien dependiendo del tipo de programa. Por ejemplo, si un programa tiene muchos bucles, el *taken* puede acertar bastante, mientras que si tiene muchos condicionales cortos, el *not taken* puede hacerlo mejor.

En resumen, los predictores estáticos no usan memoria ni tablas y no se adaptan al comportamiento real del programa, pero son útiles para entender cómo la predicción de saltos afecta al rendimiento y para tener una referencia de base antes de pasar a los predictores dinámicos.

Resultados



Para esta primera tabla cabe destacar una cosa: el Simplescalar simula exactamente igual el *taken* y el *not taken* (mismos resultados). Esto se debe a un error del simulador. Por ende, lo que hemos hecho ha sido asumir que lo que hace el Simplescalar es taken y para obtener el not taken le hemos restado a 1 el resultado del taken:

P(not taken) = 1 - P(taken)

En los predictores estáticos se ve claramente la diferencia entre asumir siempre *not taken* o siempre *taken*. El *not taken* acierta bastante más en todos los benchmarks, con una media de acierto del **77 %**, mientras que el *taken* se queda muy por detrás, con solo un **22 %**.

Esto tiene sentido, porque muchos saltos en los programas suelen ser condicionales que no se toman (por ejemplo, comprobaciones que no siempre entran en un *if* o finales de bucles), así que el *not taken* suele acertar más veces. Aunque esto no tiene que ser así siempre necesariamente, ya que si un programa cuenta con muchos bucles, elegir *taken* podría ser mucho más favorable.

El perfect no requiere mucho comentario, es un escenario idílico en el que siempre se acierto, por ende tiene un 100% de acierto.

En general, estos resultados confirman que, aunque los predictores estáticos son muy simples, el *not taken* ya nos da una base decente desde la que mejorar con los predictores dinámicos.

PREDICTORES BIMODALES

Introducción

Los predictores bimodales ya son un paso adelante respecto a los estáticos. En lugar de asumir siempre lo mismo, este tipo de predictor aprende del comportamiento pasado de cada salto. Lo que hace es guardar, en una tabla llamada Pattern History Table (PHT), un pequeño contador saturado de 2 bits para cada instrucción de salto.

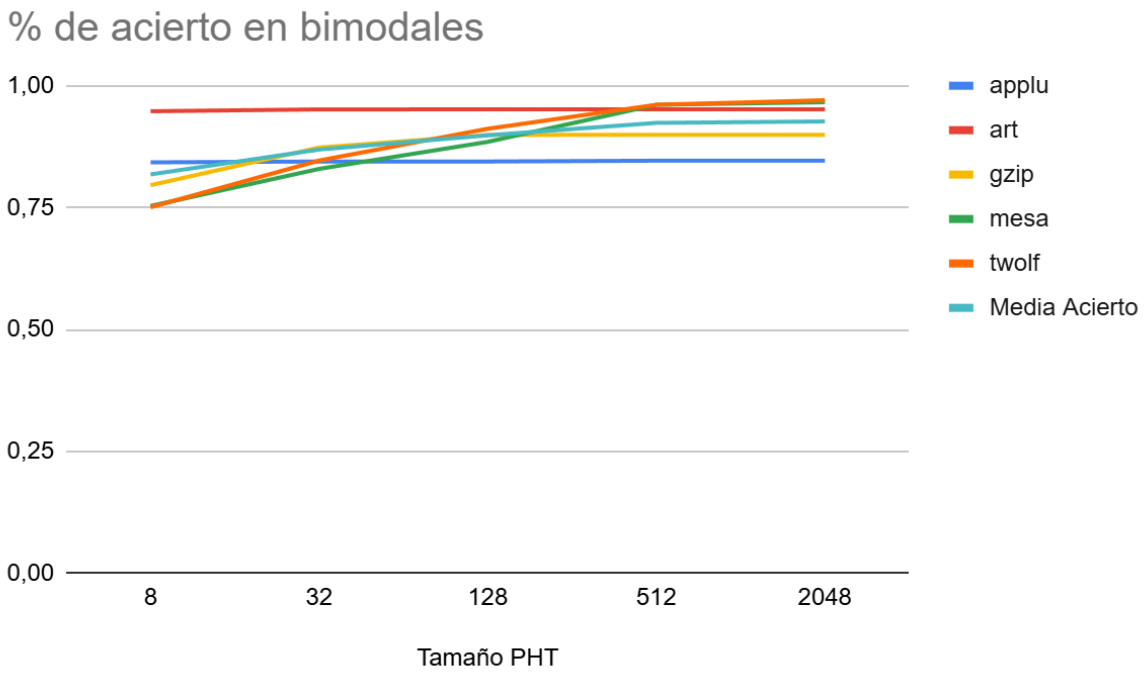
El hecho de tener un contador saturado de 2 bits hace que no cambie la predicción por fallos esporádicos, es decir, tiene un comportamiento suavizado en el que se necesitan 2 fallos seguidos para “cambiar de opinión”.

Estos contadores saturados cambian según lo que haya pasado las últimas veces: si el salto se ha tomado varias veces seguidas, el predictor tenderá a predecir *taken*, si no, *not taken*. De esta forma, el procesador no va completamente a ciegas como antes, sino que va ajustando sus predicciones en función de la experiencia.

El resultado es un predictor mucho más inteligente, capaz de adaptarse a patrones simples (por ejemplo, bucles o condicionales que se repiten de forma parecida). Además, como cada salto tiene su propio “mini historial”, evita que unos afecten a otros.

En resumen, el bimodal usa memoria y se autoajusta a medida que ejecuta el programa. No es el más avanzado, pero ya ofrece una mejora muy grande respecto a los predictores estáticos y sirve como punto de comparación para los que incorporan historia global (como Gshare o Gag).

Resultados



En los bimodales se nota claramente cómo el tamaño de la tabla (PHT) influye en la precisión del predictor. A medida que aumentamos el tamaño, el porcentaje de acierto sube de forma bastante constante, pasando de un 81,8 % con 8 entradas hasta un 92,6 % con 2048.

Esto tiene sentido, porque una tabla más grande puede almacenar más información sobre el comportamiento de cada salto, evitando que diferentes instrucciones compartan la misma entrada (*aliasing*).

Si miramos los benchmarks individualmente, art es el que siempre da mejores resultados (rozando el 95 % en todos los casos), mientras que gzip y mesa son los que más mejoran al aumentar la tabla. Twolf también escala muy bien, llegando casi al 97 % con el tamaño máximo.

En general, los resultados muestran que el predictor bimodal aprende bastante bien los patrones repetitivos de los programas, y que ampliar la PHT mejora la precisión, aunque llega un punto (entre 512 y 2048) donde la mejora ya es poco notable.

* Hemos buscado el motivo de que art tenga tan buena predicción siempre:

<<art tiene tan buena predicción porque es un programa con patrones de control muy regulares: muchos bucles y saltos que se repiten siempre igual.>>

PREDICTORES GSHARE

Introducción

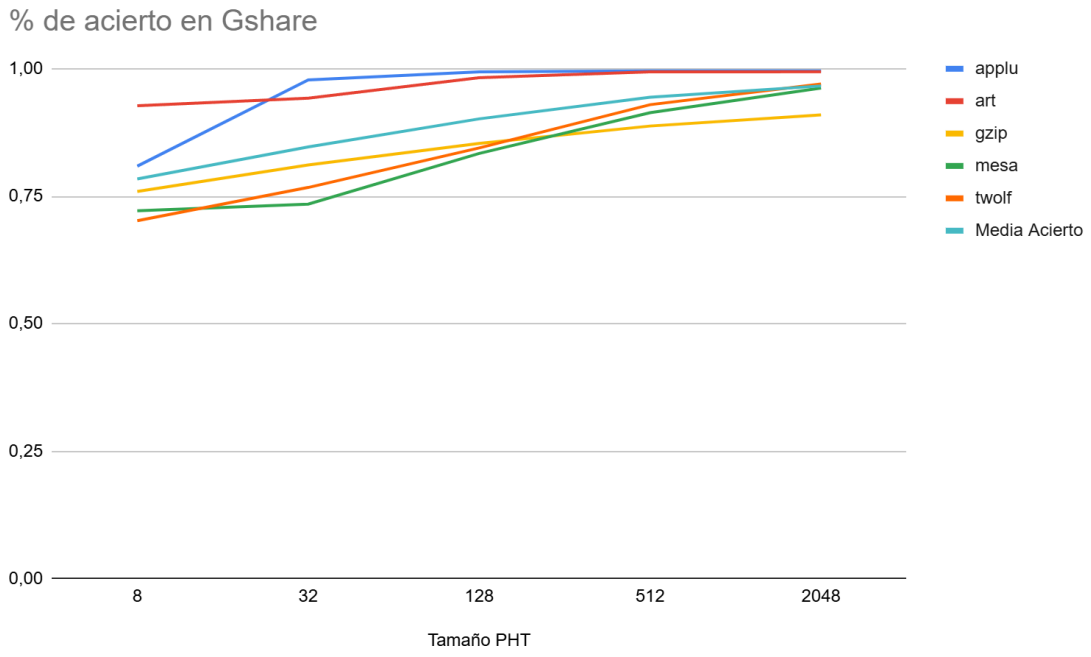
El Gshare es uno de los predictores dinámicos más conocidos y utilizados. Su idea principal es combinar la historia global de los saltos con la dirección de la instrucción actual para hacer la predicción.

Mientras que el bimodal solo miraba el comportamiento de cada salto de forma individual, el Gshare tiene en cuenta qué ha pasado en los últimos saltos del programa (por ejemplo, si los anteriores fueron *taken* o *not taken*). Esta historia global se guarda en un registro llamado GBHR (Global Branch History Register).

Luego, *para decidir qué entrada de la tabla usar (la PHT), el predictor mezcla los bits del GBHR con los bits de la dirección del salto usando una operación XOR.*

En resumen, el Gshare es más inteligente que el bimodal porque aprende patrones a nivel global, no solo por instrucción. Es un buen equilibrio entre complejidad y precisión, y suele ofrecer muy buenos resultados cuando se elige bien el tamaño de las tablas. De hecho, lo comentaremos más a fondo posteriormente, pero el Gshare con 2048 entradas es el predictor con la media de acierto más alta de todos.

Resultados



En los Gshare se nota una mejora muy clara respecto a los bimodales. Al aumentar el tamaño de la tabla (PHT), el porcentaje de acierto sube de forma casi perfecta, pasando del 78 % con 8 entradas a un 96 % con 2048. Además, el IPC también mejora bastante, lo que demuestra que el procesador aprovecha mejor los ciclos al tener menos errores de predicción.

Esto pasa porque el Gshare no solo mira cada salto por separado, sino que también tiene en cuenta lo que ha pasado en los saltos anteriores. Esa mezcla entre la historia global y la dirección de cada salto hace que el predictor aprenda patrones más complejos, por eso su precisión crece tanto con tablas grandes.

Si miramos los benchmarks, applu y art son los que más se acercan a la perfección, mientras que gzip, mesa y twolf también mejoran mucho a medida que crece el tamaño.

En resumen, el Gshare es un salto de calidad bastante notable respecto al bimodal: aprovecha la información global, escala muy bien con el tamaño de la PHT y logra un rendimiento mucho más alto, tanto en aciertos como en IPC.

Cabe destacar de nuevo que el Gshare con 2048 entradas es el que mejor resultado nos ha dado tanto en media de acierto como en media de IPC.

PREDICTORES GAG

Introducción

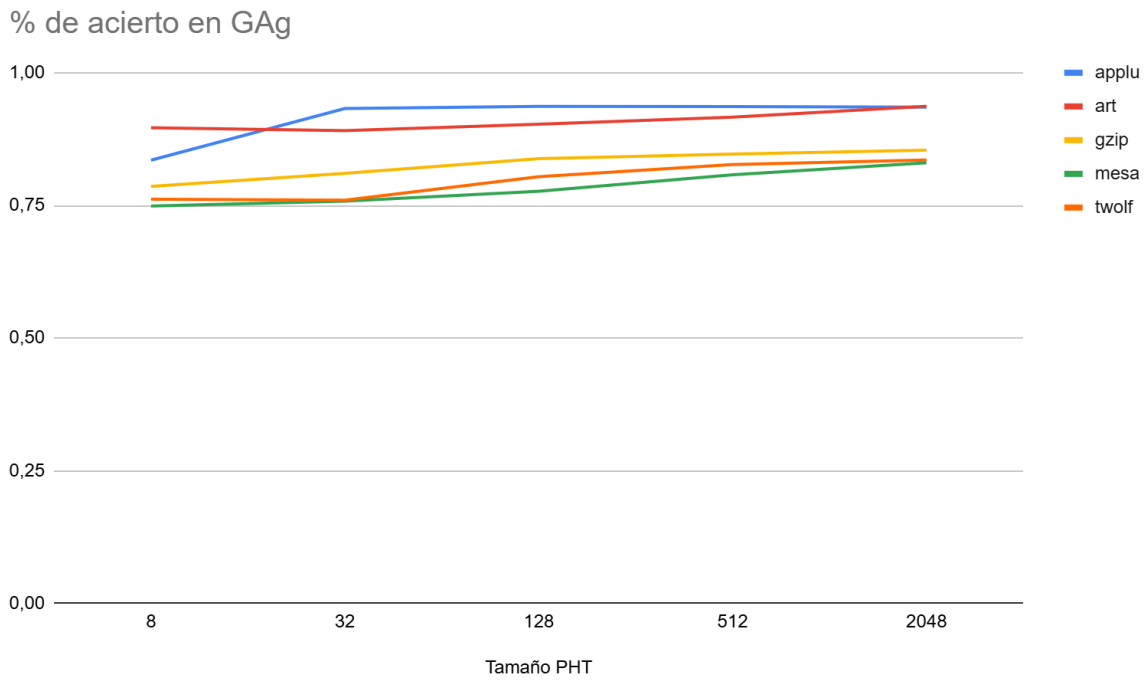
El Gag, también llamado Gselect, es bastante parecido al Gshare, pero con una pequeña diferencia en cómo mezcla la información.

La idea sigue siendo la misma: aprovechar la historia global de los saltos para mejorar las predicciones. Pero, al no mezclar los bits de BHR con los de la dirección de salto (no se concatenan, no se hace XOR…), el Gag mantiene la información más “pura”, lo que a veces ayuda y otras veces no, dependiendo de si los saltos tienen comportamientos muy parecidos o no.

En general, el Gag suele comportarse de forma muy parecida al Gshare, aunque en algunos casos puede tener un poco más de aliasing.

En resumen, el Gag es otro predictor dinámico que también utiliza la historia global, pero con una forma distinta de combinarla.

Resultados



En el caso del Gag, los resultados son buenos, pero se nota que no llega al nivel del Gshare. La media de acierto mejora un poco al aumentar el tamaño de la PHT, pasando de un 80,5 % con 8 entradas a un 87,8 % con 2048, pero la subida es más suave y no tan pronunciada como antes.

Esto tiene sentido porque el Gag no mezcla los bits del historial global con la dirección del salto (como hacía el Gshare), sino que los concatena tal cual, y eso provoca que haya más aliasing (más colisiones en la tabla). Es decir, algunas entradas se pisan entre sí y el predictor se “confunde” más.

Aun así, sigue siendo un predictor bastante sólido. Applu y art mantienen valores altos, mientras que mesa y twolf son los que más mejoran a medida que la tabla crece.

En general, el Gag demuestra que usar historia global mejora mucho respecto a los predictores más simples, pero también que la forma de combinar esa información importa mucho: el XOR del Gshare acaba siendo más efectivo.

PREDICTORES PAG

Introducción

El Pag (Per-Address Global) es otro tipo de predictor dinámico, pero en este caso cambia la forma de usar la historia. En lugar de tener una única historia global compartida por todos los saltos (como hacían el Gshare y el Gag), el Pag mantiene un historial diferente para cada instrucción de salto.

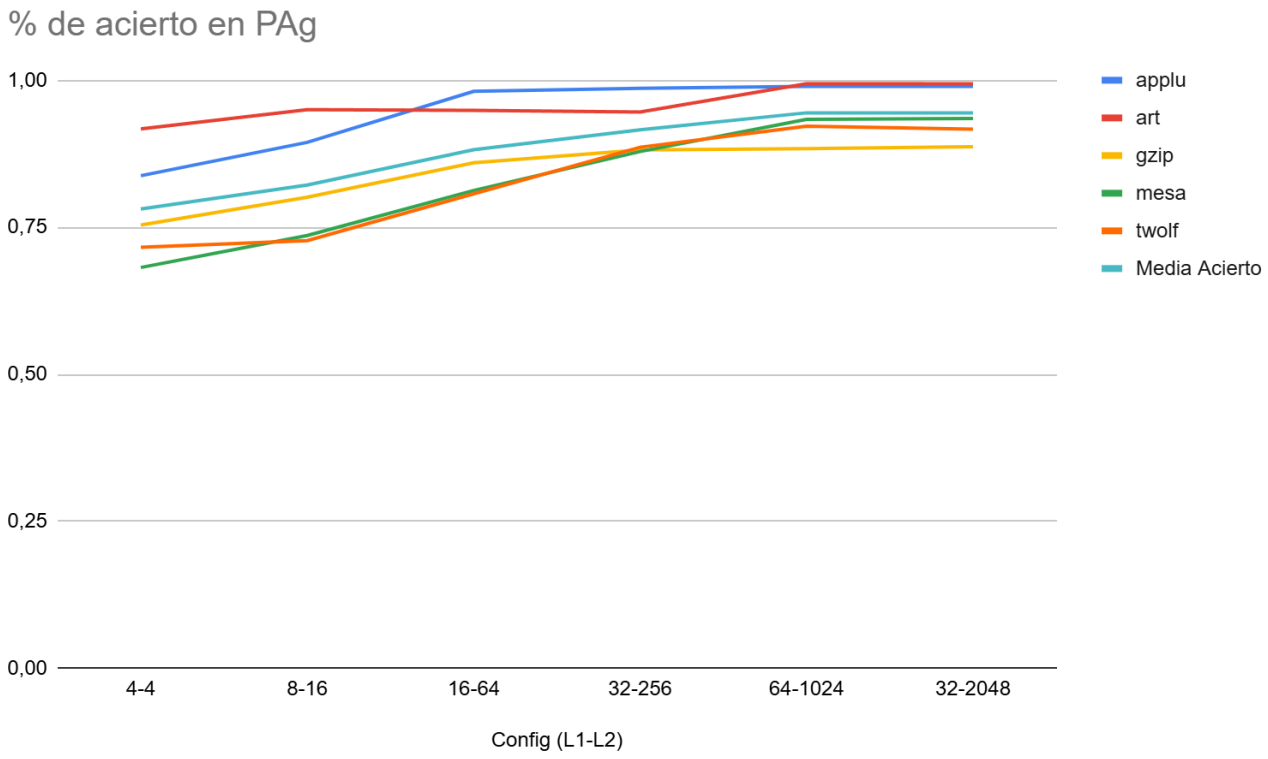
Esto se guarda en una tabla llamada PaBHT, donde cada salto tiene su propio mini registro de historia. Luego, ese historial se usa junto con una tabla de patrones (PHT) para hacer la predicción.

La gran ventaja del Pag es que aprende el comportamiento de cada salto de forma individual, sin que unos afecten a otros. Por eso suele ser muy bueno cuando los saltos del programa no siguen un patrón común, es decir, cada uno hace lo suyo.

Eso sí, al tener que guardar varias historias distintas, consume más memoria y puede ser un poco más lento de actualizar.

En resumen, el Pag combina lo mejor de los dos mundos: mantiene memoria por salto (como si cada uno tuviera su propio predictor) y, al mismo tiempo, usa una tabla global de patrones para decidir la predicción. Es más caro en recursos, pero mucho más preciso en ciertos programas.

Resultados



El predictor Pag se comporta muy bien y muestra una mejora constante a medida que aumentamos el tamaño de sus tablas (L1 y L2). Empieza con una media de acierto del 78 % en la configuración más pequeña (4-4) y llega hasta casi un 94,5 % en las configuraciones grandes (64-1024 y 32-2048).

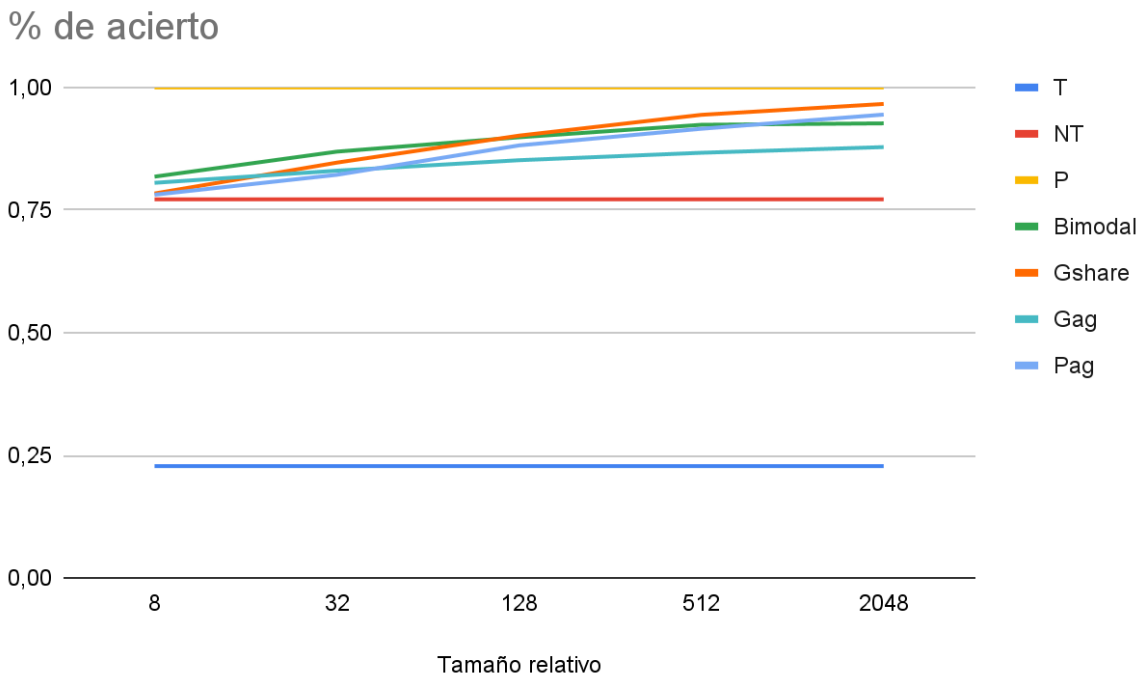
Esto demuestra que darle a cada salto su propio historial funciona muy bien, ya que el predictor aprende de forma independiente el comportamiento de cada uno, sin que unos afecten a otros.

Si miramos los benchmarks, applu y art vuelven a ser los que más destacan, alcanzando casi la perfección, mientras que gzip, mesa y twolf mejoran muchísimo conforme se amplían las tablas.

También se nota que a partir de cierto punto (cuando llegamos a 1024 o 2048 entradas), la mejora se estabiliza, lo que indica que el predictor ya tiene suficiente espacio para guardar bien las historias sin tanto aliasing.

En resumen, el Pag aprende de forma más específica y precisa, y se nota en los resultados: consigue un equilibrio muy bueno entre complejidad y precisión, superando claramente a los predictores globales como el Gshare y el Gag en la mayoría de casos.

RESULTADOS GENERALES



Si miramos todos los predictores juntos, se ve claramente la evolución de rendimiento a medida que el predictor se hace más “inteligente”.

Los estáticos (taken y not taken) marcan la base: el *taken* apenas llega al 22 %, mientras que el *not taken* se mantiene en un 77 %, lo cual no está tan mal teniendo en cuenta que no aprende nada.

A partir de ahí, los dinámicos mejoran muchísimo. El bimodal ya empieza con potencia ya que empieza con un 81 % y llega hasta más del 92 %, demostrando que aprender el comportamiento individual de cada salto marca una gran diferencia.

El Gshare destaca por encima de todos los demás clásicos, subiendo hasta casi un 97 % de acierto. Esto confirma que combinar la historia global con la dirección del salto (mediante XOR) funciona muy bien y reduce bastante los errores.

El Gag, en cambio, se queda un poco por detrás (alrededor del 87 % máximo).

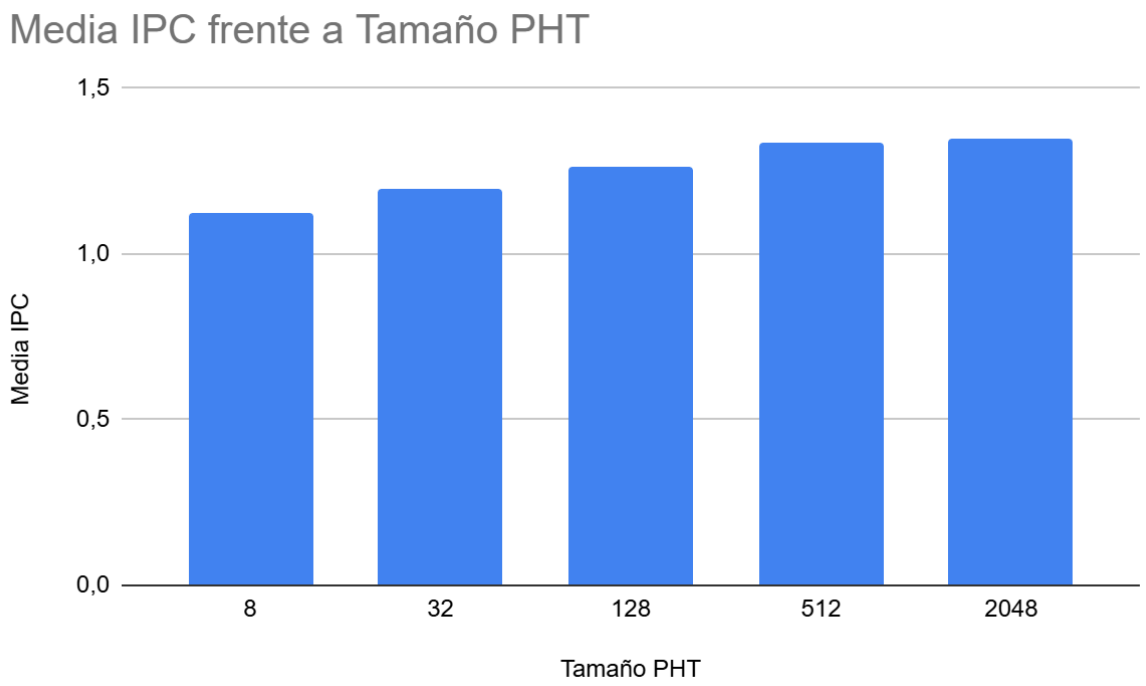
Y el Pag ofrece un rendimiento excelente, con más del 94 % en las configuraciones grandes, gracias a que mantiene un historial independiente para cada salto, evitando interferencias entre ellos.

En general, los resultados muestran que cuantos más recursos y “memoria” damos al predictor, mejor aprende el comportamiento real del programa. El salto de calidad del Gshare y el Pag respecto al resto es bastante claro, acercándose bastante al comportamiento del predictor perfecto.

3. RENDIMIENTO (IPC) SEGÚN PREDICTOR

De los predictores estáticos no hemos hecho tabla de IPC debido al error que tiene Simplescalar. Para saber la media de acierto hemos hecho la resta explicada anteriormente, pero para saber el IPC no tenemos forma alguna de saberlo, así que no lo adjuntamos ya que tanto *taken* como *not taken* son iguales, y no tiene sentido que tengan el mismo IPC si uno falla mucho más que el otro.

IPC en bimodales

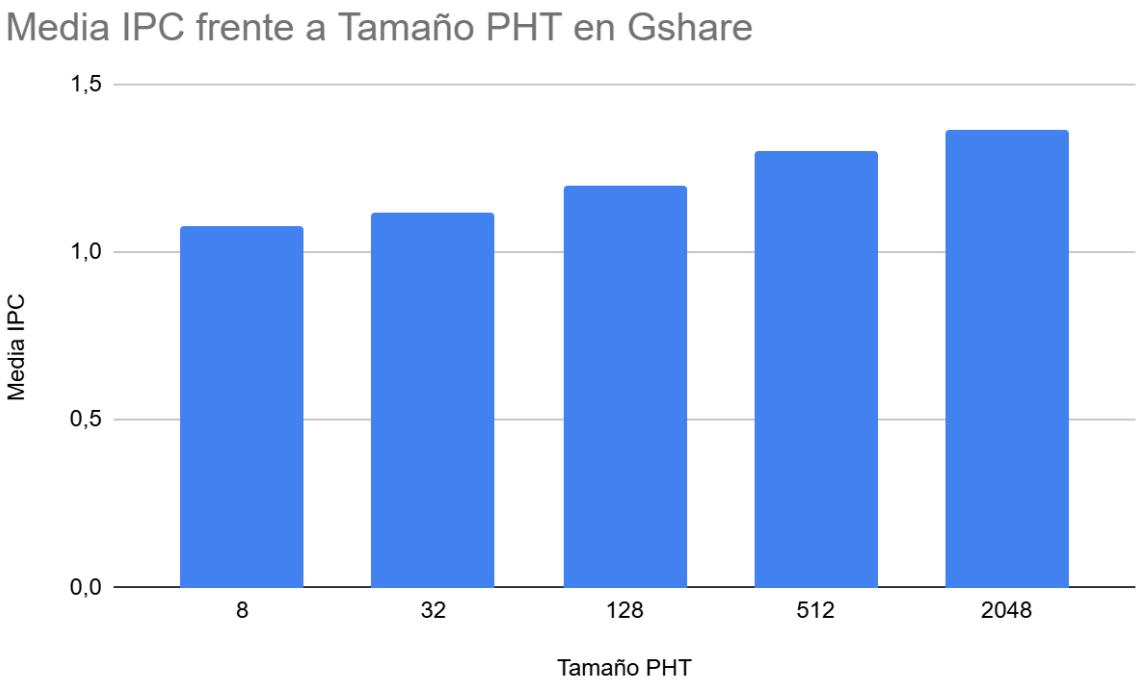


En los bimodales se ve una mejora muy clara en el rendimiento del procesador a medida que crece el tamaño de la tabla (PHT). El IPC pasa de 1,12 con 8 entradas a 1,34 con 2048, es decir, cerca de un 20% de aumento, lo que significa que el procesador consigue ejecutar más instrucciones por ciclo gracias a tener menos fallos de predicción.

Esto pasa porque, con una PHT más grande, el predictor puede aprender mejor los patrones de salto y equivocarse menos. La mejora se mantiene bastante constante hasta los tamaños grandes, donde ya empieza a estabilizarse y pasar de 512 a 2048 tiene un cambio casi imperceptible a nivel visual.

En resumen, los resultados muestran que los predictores bimodales mejoran mucho el IPC respecto a los estáticos, y que aumentar el tamaño de la PHT tiene un impacto directo en el rendimiento, aunque llega un punto donde ya no compensa tanto seguir aumentando.

IPC en Ghsare

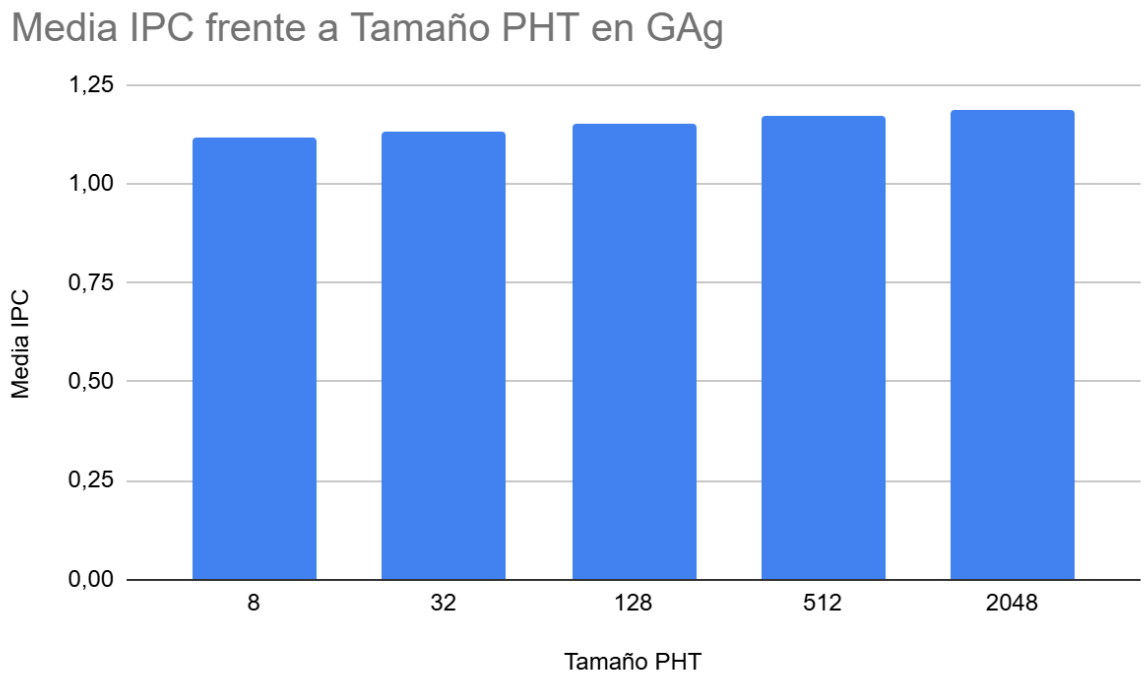


El comportamiento del Gshare es muy parecido al de los bimodales, pero con una ligera mejora general. El IPC va subiendo de forma progresiva, desde 1,07 con 8 entradas hasta 1,36 con 2048 (+27%), lo que demuestra que el predictor aprovecha mejor el paralelismo y reduce más los fallos de salto.

Esto tiene sentido, ya que el Gshare no solo usa la dirección del salto, sino también la historia global de los saltos anteriores, lo que le permite acertar más en programas con patrones repetitivos o correlaciones entre saltos.

En resumen, el Gshare aumenta el IPC a medida que crece la PHT, igual que el bimodal, pero logra un rendimiento un poco mejor gracias a su forma más avanzada de combinar la información global. De hecho, el valor máximo de IPC medio obtenido en todas las simulaciones se obtiene en el Gshare formado por 2048 entradas.

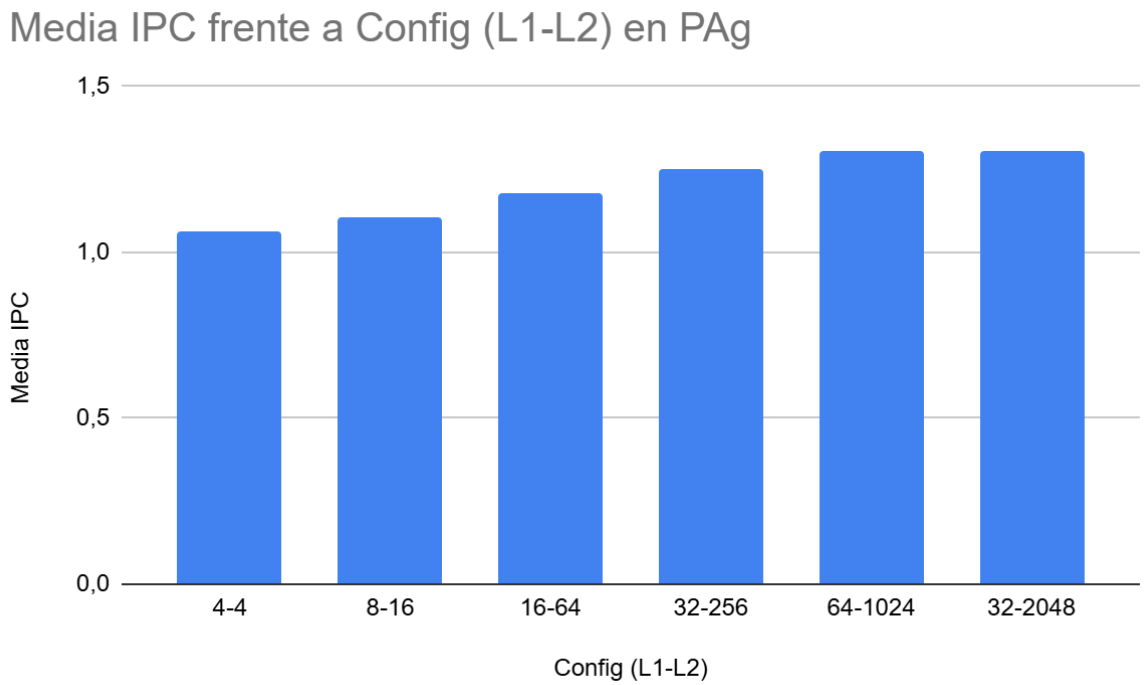
IPC en GAg



En los Gag se observa un aumento progresivo y constante del IPC al crecer el tamaño de la PHT, pasando de 1,11 con 8 entradas a 1,18 con 2048, pero muy paulatino, prácticamente imperceptible a nivel visual, con una mejora porcentual de aproximadamente 6%.

El Gag no combina tan bien la información global y la dirección del salto, lo que hace que tenga algo más de aliasing y, por tanto, un poco más de errores de predicción.

IPC en PAg



En los predictores Pag se ve una mejora constante del IPC a medida que aumentamos el tamaño de las tablas L1 y L2. Pasamos de un 1,06 con la configuración más pequeña (4-4) hasta un 1,30 con las más grandes (64-1024 y 32-2048), lo que supone un aumento del 22%.

Esto demuestra que, al tener una historia independiente para cada salto, el procesador predice mejor. Aun así, se ve que entre las dos configuraciones más grandes la mejora ya se estanca un poco, lo que indica que a partir de cierto punto el predictor ya tiene memoria suficiente para aprender el comportamiento de la mayoría de los saltos y nos demuestra que no necesariamente más tiene que ser mejor.

En resumen, el Pag ofrece un rendimiento muy alto y un IPC bastante superior al de los Gag y bimodales, mostrando que tener historia por dirección realmente ayuda a mejorar el aprovechamiento del procesador.

4. ENTREGA DE TROFEOS

Hemos tenido la graciosa idea de hacer una especie de entrega de trofeos para premiar a los predictores más destacados, y hacerlo de una forma más apetecible para el ojo.

Para determinar a los mejores nos hemos fijado en los valores medios contando a los 5 benchmarks, no valores sueltos de benchmarks puntuales.

PREMIO AL PREDICTOR CON MEJOR PRECISIÓN (% DE ACIERTO):

Gshare 2048 entradas → 96’6%

Pag 64-1024 → 94’46%

PREMIO AL PREDICTOR CON MEJOR IPC:

Gshare 2048 entradas → 1’36

Pag 64-1024 → 1’306

PREMIO AL FAIL DEL AÑO:

Taken → 22’81%

5. OBSERVACIONES FINALES

- Comparación entre predictores y comportamiento de los benchmarks

Algo que hemos visto es que no todos los benchmarks se comportan igual. El que nos llama la atención especialmente es art. Art tiene, en todo momento, un porcentaje de acierto muy alto, sin prácticamente importar el tipo de predictor y configuración. Applu sigue esta senda aunque no de la misma manera que art. Esto nos indica que los saltos siguen patrones regulares y repetitivos, lo cual resulta bastante fácil de aprender. Art trabaja mucho con matrices y arrays, es decir, es más “predecible”, por ende los predictores pueden aprender rápido.

Por otro lado, el resto de benchmarks tienen más variaciones entre predictores. Suponemos que tienen flujos más impredecibles, eso hace que sea más complicado aprender sus patrones, y por eso con configuraciones pequeñas rinden menos.

Vemos que para que un predictor sea efectivo no solo hay que tener en cuenta su complejidad, también al tipo de datos que se exponen.

- Rendimientos decrecientes

A partir de cierto punto, se empiezan a llegar al límite del crecimiento y se entra, o casi, en la parte de los rendimientos decrecientes.

Muchos predictores experimentan beneficios marginales, prácticamente nulos, y en algunos casos decreciente, al pasar de 512 a 2048 entradas. Esto nos indica que no hay que añadir todo lo más grande a lo loco, a veces hay que parar y pensar si el coste adicional en hardware y energía nos vale la pena para mejorar mínimamente.

- Aliasing

Uno de los factores clave que explican las diferencias entre predictores es el aliasing, que ocurre cuando varios saltos comparten la misma entrada en la tabla de historia.

No solo importa el tamaño del predictor, sino cómo se usa esa información.

- Conclusión personal

Nos parece asombroso como algo tan simple y pequeño puede generar un impacto tan positivo. Primero tenemos un predictor *taken* con 0’76 de IPC y al pasar a un predictor algo más avanzado como el Gshare o el PAg somos capaces de casi duplicar el rendimiento.

6. IMPLEMENTACIÓN DE PREDICTOR DE SALTOS ALLOYED

Antes de hablar de cómo hemos hecho para implementar este predictor, vamos a introducirlo y explicar en qué consiste.

El predictor Alloyed mezcla historia global y local por dirección. Se hacen 3 predicciones por separado: una local, una global y otra mixta. Para decidir si un salto es taken o not taken lo que se hace es una mayoría de los 3 resultados, y la predicción que gane es la que se ejecuta.

Para poder hacer esto, el Alloyed tiene los siguientes elementos:

- **GBHR** (formada por g bits) = array de historia global que guarda los últimos g saltos. Cuando se ejecuta un taken, se guarda un 1 en la tabla. Cuando se ejecuta un not taken, se guarda un 0.
  Si por ejemplo los últimos 3 resultados han sido todos taken, se guarda lo siguiente: 0b111.
  Cada vez que hay un nuevo salto, se desplazan los bits que ya había guardados y añades al final el nuevo.

- **PaBHT.** Es una tabla de historia local que trata de ver el comportamiento que ha tenido una instrucción de saltos. Si el parámetro k = 10, la tabla tendrá 2^10 entradas.

  El funcionamiento es el siguiente:
  Todas las instrucciones de un programa tienen una dirección (PC). No se pueden guardar todas las instrucciones (por razones obvias, hay millones). Se cogen los k bits bajos de la instrucción y se utilizan como índice de la tabla. Es como si fuera una especie de tabla de hash. El valor que se guarda es el resultado de los últimos p resultados (1 o 0).

  Por ejemplo: si p = 3, y la entrada es 0b101, quiere decir que los últimos 3 saltos de esa instrucción concreta han sido taken, not taken y taken. Si por lo que sea los k bits bajos de una instrucción coinciden con los de otra, se sobreescribe. Es decir, el predictor mira los bits bajos y luego mira ese índice en la tabla, no sabe si se trata de un beq o un bne u otra, es lo que se conoce como aliasing → dos instrucciones caen en la misma posición.

  Esta tabla ayuda para bucles y patrones repetitivos. Si el programa consiste en saltos aleatorios o en “desorden”, esta tabla pierde eficiencia.

- **PHT.** Cada entrada de la PHT es un contador saturado de dos bits:
  00 = not taken fuerte.
  01 = not taken débil
  10 = taken débil

11 = taken fuerte
Cada vez que se sabe el resultado real del salto, se actualiza el contador saturado (si es taken, se incrementa, si no lo es, se decrementa, moviéndose por estos 4 estados).

Tenemos 3 PHT distintas en nuestro predictor.

- pht_global = usa la GBHR para aprender patrones globales del programa.
- pht_local = usa la PaBHT para aprender el comportamiento de cada salto
- pht_combinada = usa GBHR y PaBHT

Para cada salto, se calculan 3 índices:

- índice global = se concatenan i bits de PC con g bits de GBHR
- índice local = se concatenan i bits de PC con p bits de PaBHT
- índice combinado = se concatenan i bits de PC con g bits de GBHR con p bits de PaBHT

(los i bits de PC son los menos significativos)

Cuando calculamos estos índices, vamos a mirar qué valor tienen asociado en sus respectivas tablas. Es como una tabla de hash: tu calculas dichos índices, con cada uno vas a su tabla pertinente y obtienes el valor que representan, es decir, su contador saturado asociado.

Con los índices calculados haces:
- pht_global[índice_global] = coges el bit más alto del contador saturado
- pht_local[índice_local] = coges el bit más alto del contador saturado
- pht_combinada[índice_combinado] = coges el bit más alto del contador saturado

Con dichos 3 bits (uno por tabla) encima de la mesa, se hace la votación.

Las tablas de GBHR y PaBHT sirven para calcular los índices con los que se accederán a las tablas PHT, ellas como tal no participan en la votación.

En resumen, el predictor Alloyed realiza tres predicciones independientes (global, local y combinada), cada una accediendo a su propia tabla de patrones (a cada una de las 3 PHT). La predicción final se obtiene mediante una votación por mayoría de los tres resultados.

Bpred.h

La primera modificación de todas es añadir la nueva clase de nuestro predictor Alloyed.

```
/* branch predictor types */
enum bpred_class {
  BPredComb,                    /* combined predictor (McFarling) */
  BPred2Level,      /* 2-level correlating pred w/2-bit counters */
  BPred2bit,       /* 2-bit saturating cntr pred (dir mapped) */
  BPredTaken,      /* static predict taken */
  BPredNotTaken,    /* static predict not taken */
  BPredAlloy,    /* Predictor Alloyed*/
  BPred_NUM
};
```

BPred_NUM tiene que ser siempre la última, ya que es la que cuenta cuántos tipos hay.

Lo siguiente que debemos hacer es crear el struct correspondiente para que desde el bpred.c se pueda crear el tipo de predictor alloy.

```
struct {
  int g;                    // número de bits del registro global (GBHR)
  int p;                    // número de bits de historia local por instrucción (PaBHT)
  int i_bits;               // bits del PC usados para completar el índice de las PHT
  unsigned int gbhr;        // registro de historia global
  unsigned int gbhr_mask;   // máscara para quedarse solo con los g bits del GBHR
  unsigned int pabht_size;  // tamaño de la tabla de historia local (PaBHT)
  int pabht_bits;           // número de bits por entrada en la PaBHT (p)
  unsigned int *pabht;      // puntero a la tabla PaBHT
  unsigned int pht_size;    // tamaño de las tablas de patrones (PHT)
  unsigned char *tabla_pht_global;  // tabla de patrones global
  unsigned char *tabla_pht_local;   // tabla de patrones local
  unsigned char *tabla_pht_combi;   // tabla de patrones combinada (global + local)
  unsigned int pht_mask;    // máscara para quedarse con los c bits del índice PHT
} alloy;
```

Hay que añadir el puntero al nuevo predictor:

```
/* 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 *meta;   /* meta predictor */
    struct bpred_dir_t *alloy;
  } dirpred;
```

Y terminamos añadiendo punteros a las 3 tablas PHT:

```
/* branch predictor update information */
struct bpred_update_t {
  char *pdir1;    /* direction-1 predictor counter */
  char *pdir2;    /* direction-2 predictor counter */
  char *pmeta;    /* meta predictor counter */
  struct {    /* predicted directions */
    unsigned int ras    : 1;  /* RAS used */
    unsigned int bimod  : 1;    /* bimodal predictor */
    unsigned int twolev : 1;    /* 2-level predictor */
    unsigned int meta   : 1;    /* meta predictor (0..bimod / 1..2lev) */
  } dir;
  /*Punteros a los contadores de cada tabla alloy creada*/
  unsigned char *alloy_global;
  unsigned char *alloy_combi;
  unsigned char *alloy_local;
};
```

Bpred.c

Para estos pasos hemos seguido paso por paso los puntos de la práctica. Lo primero que hay que hacer es añadir un nuevo case al switch para que se pueda crear el nuevo predictor Alloy (añadir en bpred_create y en bpred_dir_create):

```
/*Añadir nuevo case al switch*/
case BPredAlloy:
  pred->dirpred.alloy = bpred_dir_create(BPredAlloy, l1size, l2size, shift_width, xor);
  break;
```

En bpred_dir_create:

```
case BPredAlloy:
{
  struct bpred_dir_t *d = pred_dir;

  int pht_size = l1size;    // tamaño tabla PHT
  int pabht_size = l2size; // tamaño tabla PaBHT
  int g = shift_width;      // bits historia global
  int p = xor;              // bits historia local

  d->class = BPredAlloy;

  d->config.alloy.g = g;
  d->config.alloy.p = p;
```

```
  // Calcular i_bits = log2(pht_size) - g - p
  int c_bits = 0;
  unsigned int temp = pht_size;
  while (temp > 1)
  {
    c_bits++;
    temp >>= 1;
  }
  int i_bits = c_bits - g - p;
  if (i_bits < 0)
    fatal("PHT demasiado pequeña: se requieren al menos 2^(g+p) entradas");
  d->config.alloy.i_bits = i_bits;

  // Inicializar GBHR (todos los bits a 1)
  d->config.alloy.gbhr = (1u << g) - 1;
  d->config.alloy.gbhr_mask = (1u << g) - 1;
```

```
  // Inicializar PaBHT
  d->config.alloy.pabht_size = pabht_size;
  d->config.alloy.pabht_bits = p;
  d->config.alloy.pabht = calloc(pabht_size, sizeof(unsigned int));
  if (!d->config.alloy.pabht)
    fatal("no se ha podido reservar memoria para PaBHT");

  unsigned int inicial_local = (1u << p) - 1;
  for (int j = 0; j < pabht_size; j++)
    d->config.alloy.pabht[j] = inicial_local;

  // Inicializar las tres PHT
  d->config.alloy.pht_size = pht_size;
  d->config.alloy.pht_mask = pht_size - 1;
  d->config.alloy.tabla_pht_global = calloc(pht_size, sizeof(unsigned char));
  d->config.alloy.tabla_pht_local = calloc(pht_size, sizeof(unsigned char));
  d->config.alloy.tabla_pht_combi = calloc(pht_size, sizeof(unsigned char));

  if (!d->config.alloy.tabla_pht_global || !d->config.alloy.tabla_pht_local || !d->config.alloy.tabla_pht_combi)
  {
    fatal("no se ha podido reservar memoria para las PHT");
  }
```

```
    // Inicializar contadores (1 = not taken débil)
    for (int j = 0; j < pht_size; j++)
    {
      d->config.alloy.tabla_pht_global[j] = 1;
      d->config.alloy.tabla_pht_local[j] = 1;
      d->config.alloy.tabla_pht_combi[j] = 1;
    }
    break;
  }

  default:
    panic("error en el switch");
  }

  return pred_dir;
}
```

Después hay que añadir un nuevo case al switch de bpred_config y bpred_dir_config para poder mostrar la configuración del predictor por la salida del simulador.

```
case BPredAlloy:
  fprintf(stream,
          "pred_dir: %s: Alloy -> PHT=%u, PaBHT=%u, g=%d, p=%d, i=%d\n",
          name,
          pred_dir->config.alloy.pht_size,
          pred_dir->config.alloy.pabht_size,
          pred_dir->config.alloy.g,
          pred_dir->config.alloy.p,
          pred_dir->config.alloy.i_bits);
  break;
```

```
case BPredAlloy:
  bpred_dir_config(pred->dirpred.alloy, "alloy", stream);
  fprintf(stream, "btb: %d sets x %d associativity", pred->btb.sets, pred->btb.assoc);
  fprintf(stream, "ret_stack: %d entries", pred->retstack.size);
  break;
```

Añadimos un nuevo case en bpred_reg_stats para poder ver el resultado de la simulación:

```
case BPredAlloy:
  name = "bpred_alloy";
  break;
```

Añadimos nuevo case en bpred_dir_lookup:

```
case BPredAlloy:
{
  unsigned int gbhr = pred_dir->config.alloy.gbhr & pred_dir->config.alloy.gbhr_mask;

  unsigned int pc_index = (baddr >> MD_BR_SHIFT) & ((1 << pred_dir->config.alloy.i_bits) - 1);
  unsigned int pa_index = (baddr >> MD_BR_SHIFT) & (pred_dir->config.alloy.pabht_size - 1);
  unsigned int pa_bhr = pred_dir->config.alloy.pabht[pa_index];

  unsigned int idx_global = (pc_index << pred_dir->config.alloy.g) | gbhr;
  idx_global &= pred_dir->config.alloy.pht_mask;

  unsigned int idx_local = (pc_index << pred_dir->config.alloy.p) | pa_bhr;
  idx_local &= pred_dir->config.alloy.pht_mask;

  unsigned int idx_mixed = (pc_index << (pred_dir->config.alloy.g + pred_dir->config.alloy.p)) | (gbhr << pred_dir->config.alloy.p) | pa_bhr;
  idx_mixed &= pred_dir->config.alloy.pht_mask;

  int votos_tomado = 0;
  if (pred_dir->config.alloy.pht_global[idx_global] >= 2)
    votos_tomado++;
  if (pred_dir->config.alloy.pht_local[idx_local] >= 2)
    votos_tomado++;
  if (pred_dir->config.alloy.pht_mixed[idx_mixed] >= 2)
    votos_tomado++;

  return (char *)&pred_dir->config.alloy.tabla_pht_combi[idx_mixed];
}
```

bpred_lookup:

```
case BPredAlloy:
  if ((MD_OP_FLAGS(op) & (F_CTRL|F_UNCOND)) != (F_CTRL|F_UNCOND)) {
    struct bpred_dir_t *d = pred->dirpred.alloy;

    unsigned int gbhr = d->config.alloy.gbhr & d->config.alloy.gbhr_mask;
    unsigned int pc_index = (baddr >> MD_BR_SHIFT) & ((1 << d->config.alloy.i_bits) - 1);
    unsigned int pa_index = (baddr >> MD_BR_SHIFT) & (d->config.alloy.pabht_size - 1);
    unsigned int pa_bhr = d->config.alloy.pabht[pa_index];

    unsigned int idx_global = ((pc_index << d->config.alloy.g) | gbhr) & d->config.alloy.pht_mask;
    unsigned int idx_local  = ((pc_index << d->config.alloy.p) | pa_bhr) & d->config.alloy.pht_mask;
    unsigned int idx_mixed  = ((pc_index << (d->config.alloy.g + d->config.alloy.p))
                            | (gbhr << d->config.alloy.p) | pa_bhr)
                            & d->config.alloy.pht_mask;

    dir_update_ptr->alloy_global = &d->config.alloy.tabla_pht_global[idx_global];
    dir_update_ptr->alloy_local  = &d->config.alloy.tabla_pht_local[idx_local];
    dir_update_ptr->alloy_combi  = &d->config.alloy.tabla_pht_combi[idx_mixed];

    dir_update_ptr->pdir1 = (char *)dir_update_ptr->alloy_combi;

    int votos_tomado = 0;
    if (*dir_update_ptr->alloy_global >= 2) votos_tomado++;
    if (*dir_update_ptr->alloy_local  >= 2) votos_tomado++;
    if (*dir_update_ptr->alloy_combi  >= 2) votos_tomado++;

    dir_update_ptr->dir.bimod = (votos_tomado >= 2);
  }
  break;
```

Finalmente, hay que actualizar todas las tablas una vez se sabe el resultado exacto de la instrucción de salto (si ha sido taken o no):

```
if (pred->class == BPredAlloy && (MD_OP_FLAGS(op) & (F_CTRL|F_UNCOND)) != (F_CTRL|F_UNCOND)) {
  struct bpred_dir_t *d = pred->dirpred.alloy;

  // Actualizar los tres contadores (saturantes)
  if (dir_update_ptr->alloy_global) {
    if (taken && *dir_update_ptr->alloy_global < 3) ++*dir_update_ptr->alloy_global;
    else if (!taken && *dir_update_ptr->alloy_global > 0) --*dir_update_ptr->alloy_global;
  }

  if (dir_update_ptr->alloy_local) {
    if (taken && *dir_update_ptr->alloy_local < 3) ++*dir_update_ptr->alloy_local;
    else if (!taken && *dir_update_ptr->alloy_local > 0) --*dir_update_ptr->alloy_local;
  }

  if (dir_update_ptr->alloy_combi) {
    if (taken && *dir_update_ptr->alloy_combi < 3) ++*dir_update_ptr->alloy_combi;
    else if (!taken && *dir_update_ptr->alloy_combi > 0) --*dir_update_ptr->alloy_combi;
  }

  // Actualizar historia local (PaBHT)
  unsigned int index_pa = (baddr >> MD_BR_SHIFT) & (d->config.alloy.pabht_size - 1);
  unsigned int mask_p = (1u << d->config.alloy.pabht_bits) - 1;
  unsigned int old_pa = d->config.alloy.pabht[index_pa] & mask_p;
  unsigned int new_pa = ((old_pa << 1) | (taken ? 1u : 0u)) & mask_p;
  d->config.alloy.pabht[index_pa] = new_pa;

  // Actualizar historia global
  d->config.alloy.gbhr = ((d->config.alloy.gbhr << 1) | (taken ? 1u : 0u)) & d->config.alloy.gbhr_mask;
}
```

Sim-outorder.c

En el sim-outorder hay que añadir, en primer lugar, las variables pertinentes:

```
/* Configuración del predictor Alloy */
static int alloy_nelt = 0;
static int alloy_config[4];
```

Esto nos permitirá leer la configuración que le pasaremos por terminal.

Posteriormente, añadimos:

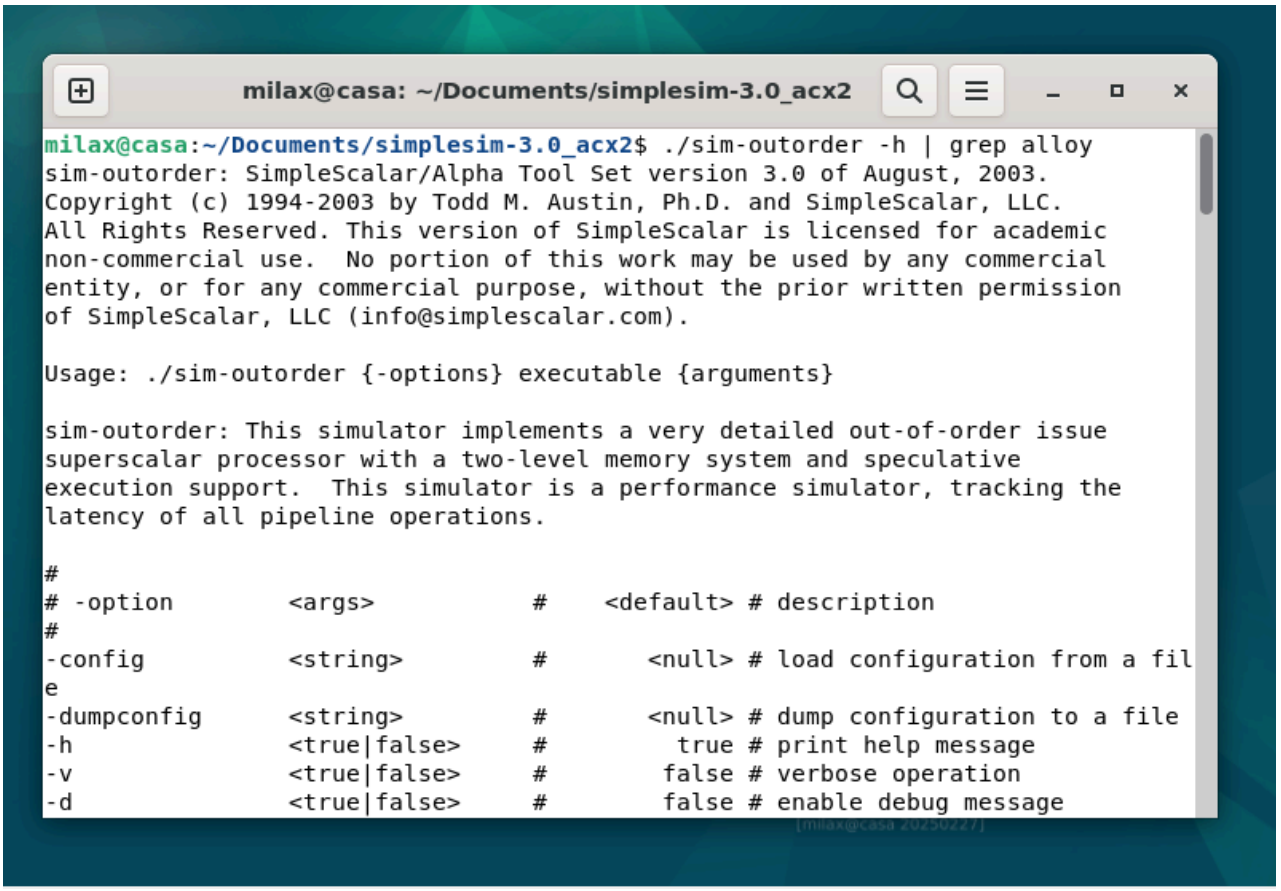
```
opt_reg_int_list(odb, "-bpred:alloy",
  "configuración predictor alloy (<pht_size> <pabht_size> <p> <g>)",
  alloy_config, 4, &alloy_nelt,
  NULL, TRUE, NULL, FALSE);
```

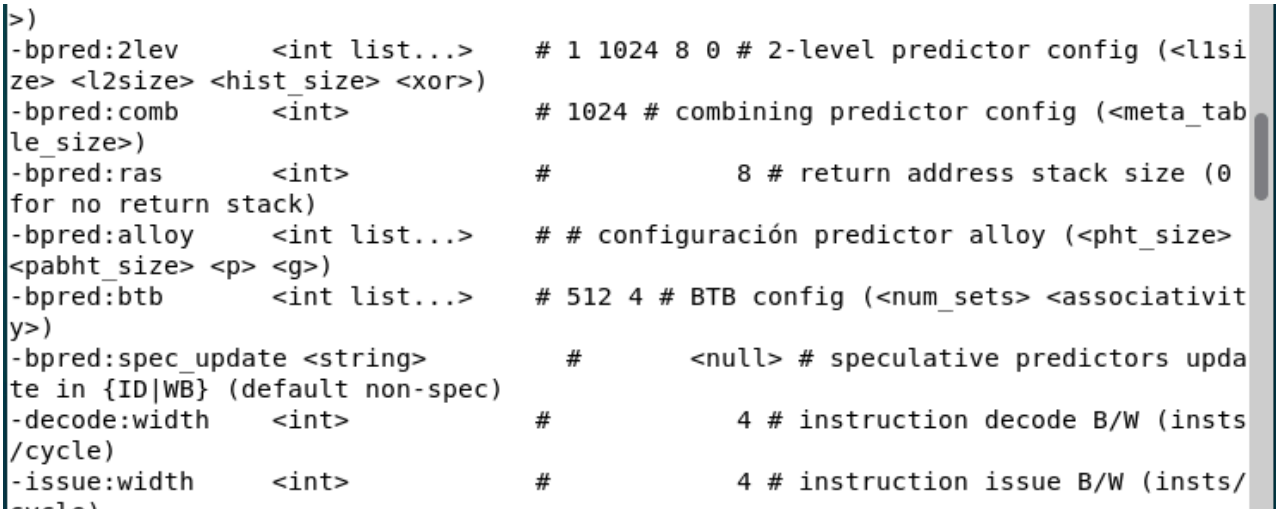
Finalmente, al igual que se crean los predictores taken, not taken, bimodales… hay que crear el nuestro:

```
else if (!mystricmp(pred_type, "alloy"))
{
if (alloy_nelt != 4)
    fatal("configuración inválida: -bpred:alloy requiere 4 parámetros (<pht_size> <pabht_size> <p> <g>)");

if (btb_nelt != 2)
    fatal("configuración inválida del BTB (<num_sets> <associativity>)");

pred = bpred_create(BPredAlloy,
                    0,                  /* bimodal table size */
                    alloy_config[0],    /* l1size = PHT size */
                    alloy_config[1],    /* l2size = PaBHT size */
                    0,                  /* meta size */
                    alloy_config[3],    /* shift_width = g bits */
                    alloy_config[2],    /* xor = p bits */
                    btb_config[0],      /* btb sets */
                    btb_config[1],      /* btb assoc */
                    ras_size);          /* return stack size */
}
```

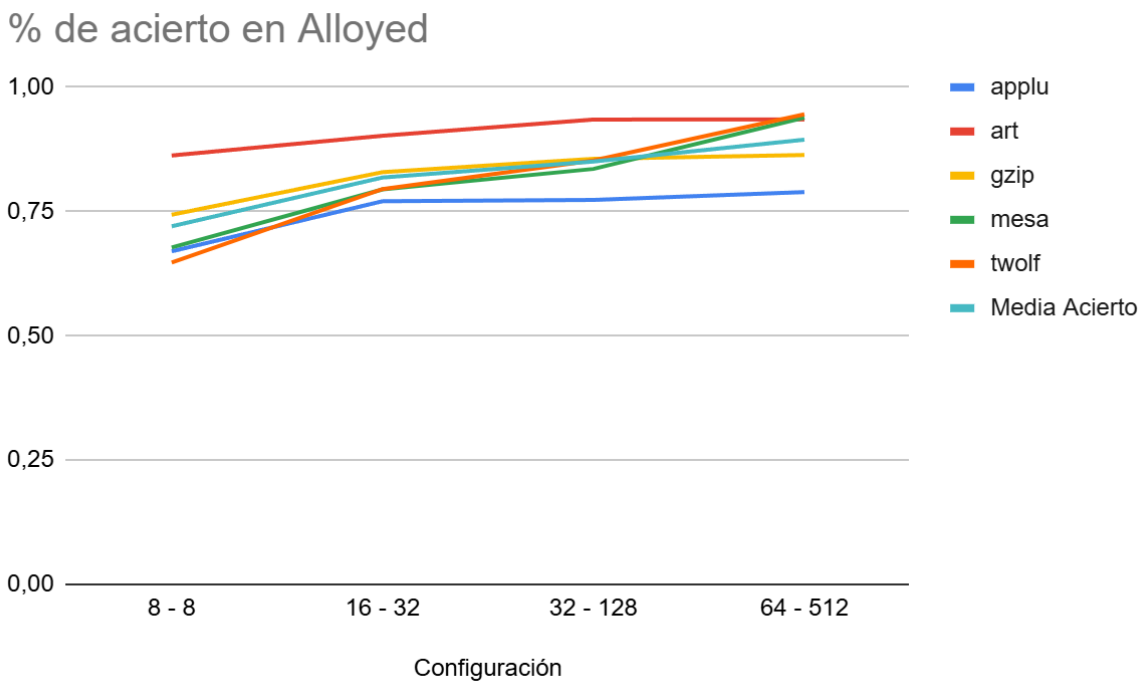
7. VALIDACIÓN ALLOYED

Para validar que realmente aparece el nuevo predictor, hemos escrito el comando: "./sim-outorder -h | grep alloy" y como se muestra en la segunda captura, aparece dicho predictor.





Hemos hecho las simulaciones para cada benchmark con las configuraciones requeridas. Los resultados son los siguientes:



En el predictor alloy se ve claramente que cuanto más grandes son las tablas, mejor funciona. La configuración más pequeña (8–8) tiene una media de acierto bastante baja, alrededor del 72%, porque prácticamente no puede capturar casi nada de historia y sufre mucho aliasing.

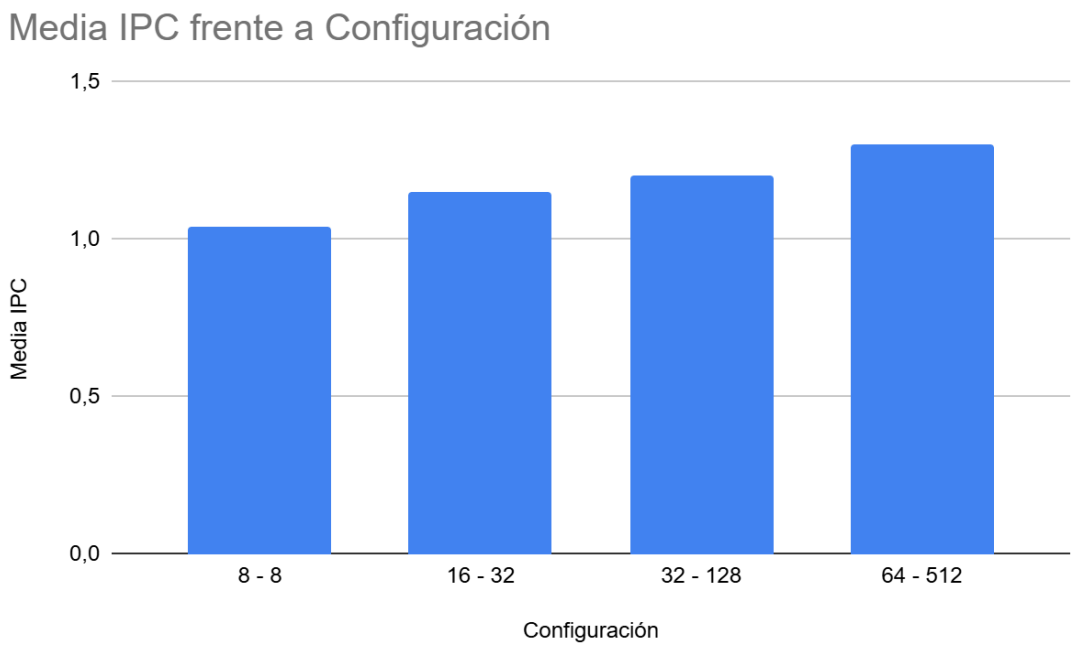
Cuando pasamos a 16–32, la mejora es muy notable: sube a un 81%, lo cual ya indica que el predictor empieza a tener más contexto y puede aprovechar mejor tanto la parte global como la local.

Con 32–128, la mejora continúa pero ya es más reducida, llegando casi al 85%. Aquí ya se ve una especie de “curva de saturación”, porque de 16–32 a 32–128 sí mejora, pero no tanto como antes.

Y finalmente, con la configuración más grande (64–512), el alloy llega a casi un 90% de acierto de media, que es un valor bastante alto. En benchmarks como mesa y twolf la mejora es espectacular, *sobre todo porque son programas que tienen bastante correlación en los saltos y se benefician mucho de tener más bits de historia*.

En resumen, el alloy mejora cuando aumentan las tablas y parece que a partir de cierto punto los beneficios ya no crecen tanto, pero igualmente la última configuración es claramente la mejor de todas.

Destacar que las configuraciones 128-2048 y 64-4096 no hemos podido hacerlas porque nos ha dado error.



En cuanto al IPC, la tendencia es prácticamente la misma que con la tasa de acierto: a mayor tamaño de las tablas del alloy, mejor rendimiento consigue el procesador (más aciertos = más rendimiento).

Con la configuración más pequeña (8–8), el IPC es de 1’03, que es bastante bajo y refleja que se están produciendo muchos fallos de predicción.

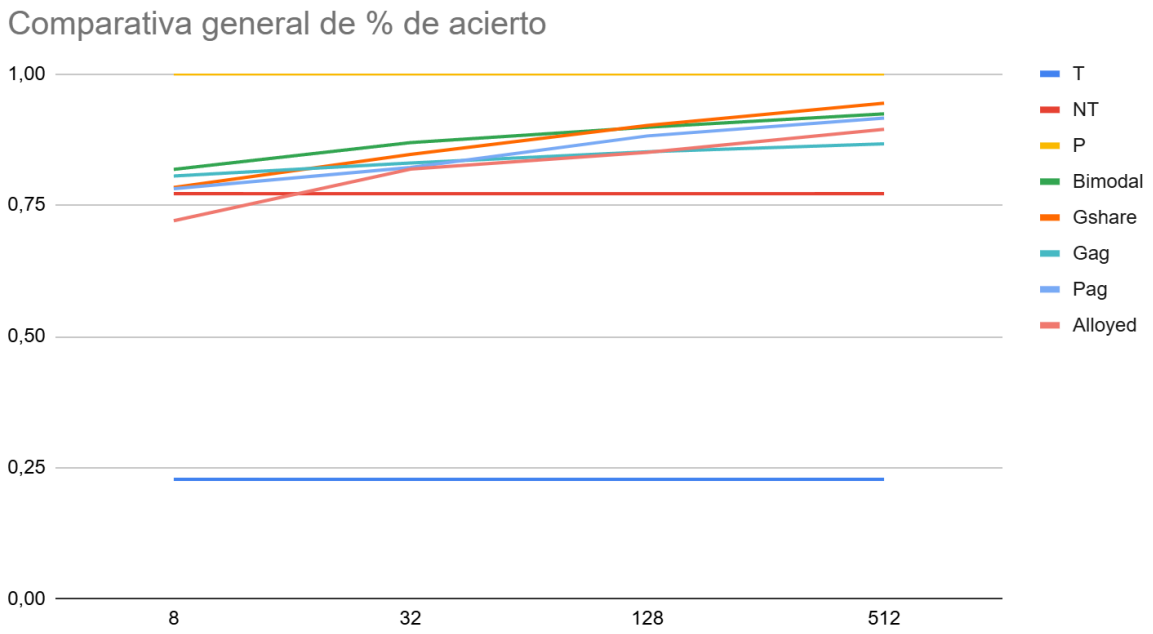
Cuando aumentamos a 16–32, el IPC sube a 1’15, lo cual encaja perfectamente con la mejora de acierto que había antes. Aquí ya se nota que el procesador funciona mejor porque fallan menos saltos.

Con 32–128, vuelve a mejorar, llegando a 1’20, pero igual que en la tasa de acierto, la mejora ya es más progresiva: el predictor ya empieza a capturar suficiente historia y los beneficios son menos bruscos.

Finalmente, con la configuración más grande (64–512), el IPC llega a 1’30, que es el mejor valor. Esto cuadra totalmente con que en esta configuración el alloy tiene la menor cantidad de fallos de predicción y por tanto aprovecha mejor el pipeline.

En resumen: más tamaño = más acierto = más IPC.

Para acabar, hemos añadido los resultados del Alloyed a la tabla de resultados de la fase 1, para hacer una comparativa con los porcentajes de acierto:



En la tabla general se ve que el alloy empieza siendo el predictor que peor funciona cuando el tamaño es pequeño. Con 8 entradas, su media de acierto es de 0’72, claramente por detrás de bimodal, gshare o gag. Esto tiene sentido porque el alloy necesita *cierta cantidad* de historia global y local para realmente combinar bien la información. Con tablas tan pequeñas sufre muchísimo aliasing.

Cuando paso al tamaño 32, el alloy mejora bastante y sube a 0’81, colocándose más cerca del resto de predictores.

En 128 entradas, el alloy sigue subiendo hasta 0’85, aunque aquí se ve que gshare y pag ya van por delante y aprovechan mejor estas tablas medianas. Aun así, la mejora del alloy es notable.

Finalmente, con 512 entradas, el alloy llega a 0’89, que es su mejor resultado y donde por fin se acerca más al rendimiento del resto de predictores avanzados. No llega a los valores de gshare o pag, pero mejora muchísimo respecto a los tamaños pequeños, mostrando claramente que es un predictor que escala bien y que necesita tablas grandes para rendir de verdad.

En resumen: el alloy empieza justito, pero mejora mucho al aumentar el tamaño y acaba con un rendimiento bastante decente, aunque sin llegar a ser el mejor entre los demás.

¿A qué otro predictor (GAg, PAg, GAp, PAp) se parece el alloy? ¿Con qué parámetros? Simulación y comparación con el alloy implementado

Por cómo está construido y por los resultados que obtiene, el predictor alloy se parece principalmente a un predictor de tipo PAg/Pag, es decir, un predictor que utiliza historia por dirección (local) combinada con historia global.

El alloy trabaja con:

- un registro de historia global (GBHR)
- un historial por dirección (PaBHT)
- y tres PHT distintas

Esto hace que, en cuanto a comportamiento, esté entre medias de un GAg (más global) y un PAg (más local), aunque por los resultados parece más parecido a PAg, sobre todo cuando los tamaños de las tablas son grandes.