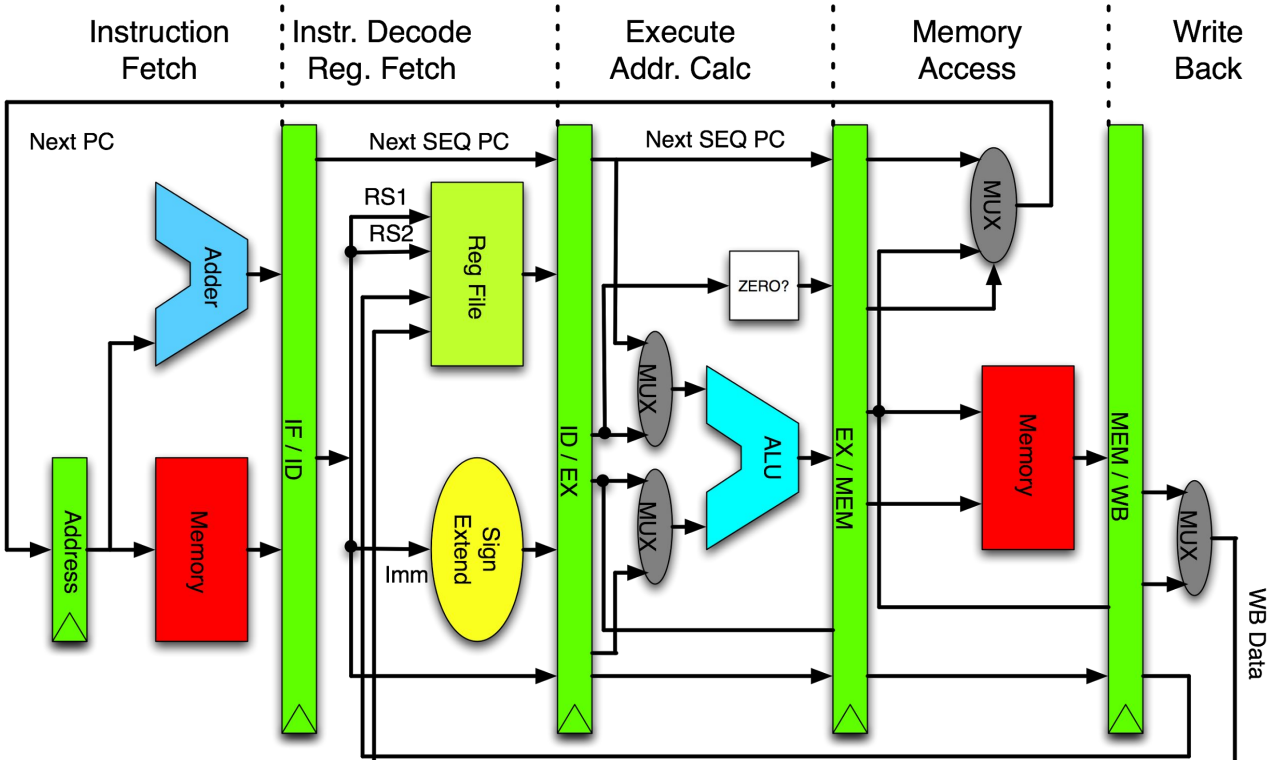
Predyktory skoków

Seminarium ASK 2017

Jakub Piecuch

- Weźmy prosty procesor potokowy, np. MIPS
- Chcemy, żeby potok w każdej chwili był wypełniony instrukcjami



Przeszkoda: instrukcje skoku

- Skok bezwarunkowy (`jal, jr, j`):
  - Zmienia adres, z którego pobierane są kolejne instrukcje
  - Czy nowy adres znany jest od razu (najlepiej w fazie pobierania)?
- Skok warunkowy (`beq, bne,` itp):
  - **Może** zmienić adres, z którego pobierane są kolejne instrukcje
  - Żeby poznać nowy adres, musimy poznać prawdziwość warunku
  - Prawdziwość warunku znana dopiero w fazie wykonania (*execute*)

Nie wszystkie skoki sobie równe

Typ skoku	**Kierunek znany w czasie dekodowania?**	**Liczba możliwych adresów następnej instrukcji**	**Kiedy znany jest adres następnej instrukcji?**
Warunkowy `beq`	**Nie**	**2**	**Faza wykonania (skok zależny od wartości rejestrów)**
Bezwarunkowy (adres w instrukcji) `j, jal`	Tak (zawsze wykonany)	1	Faza dekodowania (PC + offset)
Bezwarunkowy (adres w rejestrze) `jr`	Tak (zawsze wykonany)	**Wiele**	**Faza wykonania (skok zależny od wartości rejestrów)**

Proste rozwiązania

1. Wstrzymaj wykonanie instrukcji następujących po skoku do momentu obliczenia prawdziwości warunku (*stall*)
   - Proste w implementacji
   - Niewykorzystane zasoby, szczególnie w architekturach superskalarnych i o dłuższych potokach
2. Aby zamaskować opóźnienie wykonaj instrukcję bezpośrednio po skoku niezależnie od prawdziwości warunku / adresu docelowego (*delay slots*)
   - Większe wykorzystanie zasobów
   - Dodatkowo obciąża programistę / kompilator

Powyższe metody są niepraktyczne w nowoczesnych implementacjach

- Długości typowych potoków wahają się w granicach 10-20
  - Pentium 4 Prescott: 31
  - Skylake, Kabylake, Haswell: 14
- Nie możemy czekać kilkanaście cykli przy każdym skoku warunkowym!
- *Delay slots* też odpadają:
  - Bardzo trudno jest wypełnić większą ich liczbę
  - Liczba *delay slots* znajdujących się po każdym musi być stała w obrębie danej architektury
  - Różne implementacje tej samej architektury mogą wymagać ich mniej lub więcej

Proste rozwiązania, c.d. (statyczne przewidywanie skoków)

3. Załóż, że wszystkie wszystkie skoki warunkowe zostaną wykonane (lub nie)
   - Skoki częściej są wykonywane niż nie (pętle)
   - Jeśli się pomylimy trzeba unieważnić wszystkie instrukcje, które rozpoczęliśmy wykonywać po skoku
   - Konieczne wprowadzenie dodatkowych mechanizmów (unieważnianie instrukcji)
4. Modyfikacja metody 3: skoki do tyłu wykonane, skoki do przodu nie
   - Korzysta z typowego zachowania pętli
5. Wskazówki umieszczone przez programistę w kodzie źródłowym lub przez kompilator na podstawie analizy statycznej programu
   - Czasami programista lub kompilator wie który kierunek jest bardziej prawdopodobny

Wszystkie te rozwiązania mają wspólną wadę: nie są wrażliwe na dynamiczne zmiany w zachowaniu programu

Ciekawsze rozwiązania

- W trakcie czekania na obliczenie prawdziwości warunku wykonuj instrukcje z innego wątku (*fine-grained multithreading*)
- Prawie całkowita eliminacja instrukcji skoku warunkowego, w zamian wykonanie pojedynczych instrukcji można uzależnić od jakiegoś warunku (*predicated execution*)
  - `CMOVcc` - *conditional move* (x86)
  - Większość instrukcji w architekturze Itanium
- Wykonuj instrukcje z obu możliwych ścieżek naraz (*multi-path execution*)

Dynamiczne przewidywanie skoków

- Obserwacja: Większość instrukcji skoku warunkowego wykazuje przewidywalne zachowanie
- Idea: Przewidujemy kierunek skoku i adres docelowy na podstawie dotychczasowego działania programu
- Żeby móc załadować poprawny adres następnej instrukcji musimy wiedzieć w fazie **pobierania** (*fetch stage*):
    - Czy pobierana instrukcja jest skokiem
    - Czy ten skok się wykona (w przypadku skoków warunkowych)
    - Jaki jest adres docelowy skoku

Jak przewidzieć adres docelowy?

- Obserwacja: Dla większości skoków (nie tylko warunkowych) adres następnej instrukcji w przypadku wykonania skoku nie zmienia się
- Zapamiętaj odwzorowanie pomiędzy adresami instrukcji skoku a adresami docelowymi w podręcznej strukturze danych
- W literaturze: *Branch Target Buffer* (*BTB*) lub *Branch Target Address Cache* (*BTAC*)
- Zasada działania identyczna do pamięci podręcznej (tj. *direct-mapped* lub *set-associative*)
- Pozwala też przewidzieć czy dana instrukcja jest skokiem (*BTB hit* → tak, *BTB miss* → nie)

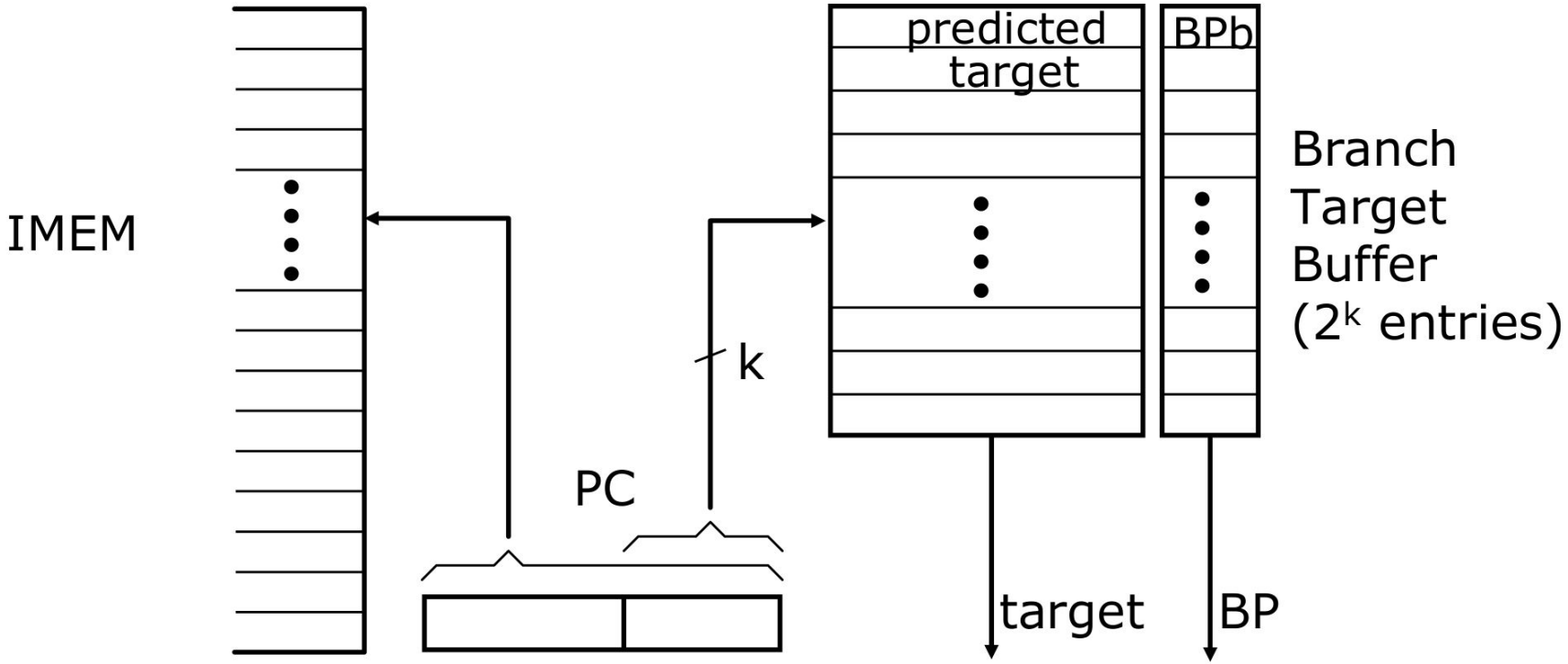
Jak przewidzieć wynik porównania?

- Musimy gdzieś trzymać informacje o “historii” wykonania danego skoku (lub wszystkich skoków) - globalna vs lokalna historia
- Najprostsze podejście: Dla każdej instrukcji skoku w BTB przechowujemy dodatkowy bit przechowujący kierunek skoku z ostatniego wykonania tej instrukcji (1 - skok został wykonany, 0 - skok nie został wykonany)
- Przewidujemy że następnym razem kierunek będzie taki sam jak ostatnio

“010101010101” → dokładność 0% (przy stanie początkowym 1)

“111111000000” → dokładność 91,7% (przy stanie początkowym 1)

1-bitowa lokalna historia - schemat



Problem: aliasing

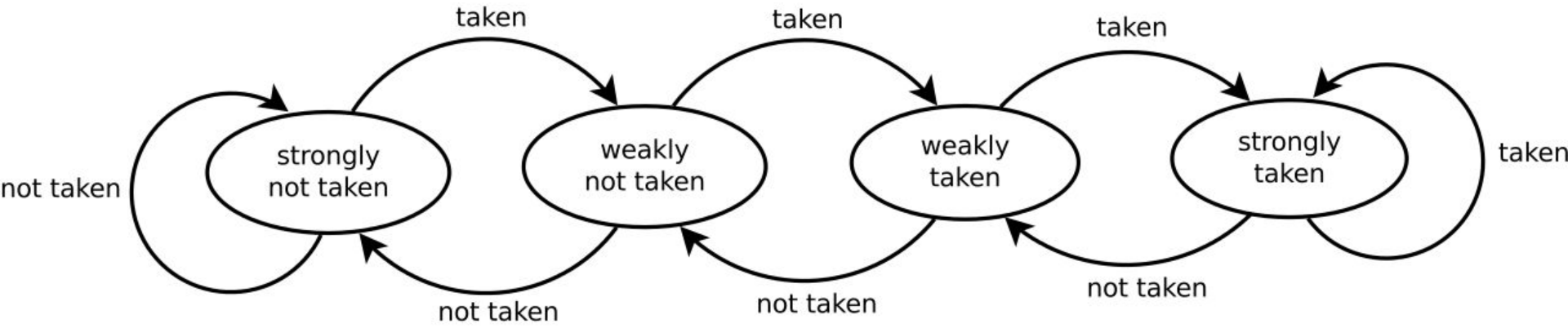
- Do adresowania BTB używamy k bitów adresu
- Istnieje wiele adresów którym odpowiada ten sam indeks w BTB
- Może się zdarzyć że instrukcja która nie jest skokiem zostanie potraktowana jako skok!
- Rozwiązanie: w BTB przechowujemy dodatkowo pozostałe bity adresu instrukcji jako znacznik (*tag*) aby móc odróżnić od siebie instrukcje pod adresami o identycznych indeksach w BTB
  - Nie jest stosowane w praktyce ze względu na ograniczenia pamięciowe
  - Zazwyczaj znacznik to zbiór bitów adresu rozłączny z bitami używanymi do indeksowania BTB
- Częściowe rozwiązanie: stosowanie pamięci sekcyjno-skojarzeniowej w BTB (podział na zbiory)

Dwa bity zamiast jednego (predyktor bimodalny)

- Założenie: większość skoków warunkowych jest “zazwyczaj wykonana” lub “zazwyczaj niewykonana”
- Dla każdego skoku zamiast 1-bitowej historii trzymamy 2-bitowy **licznik nasyceniowy** (*saturating counter*)
- Przewidywany kierunek skoku = najbardziej znaczący bit licznika
- Licznik aktualizowany po poznaniu rzeczywistego kierunku (skok wykonany → dodaj 1, skok niewykonany → odejmij 1)

“1110111011101110” → dokładność 75% (przy początkowym stanie 11 lub 10)

Predyktor bimodalny: automat skończony



Czy to wystarczy?

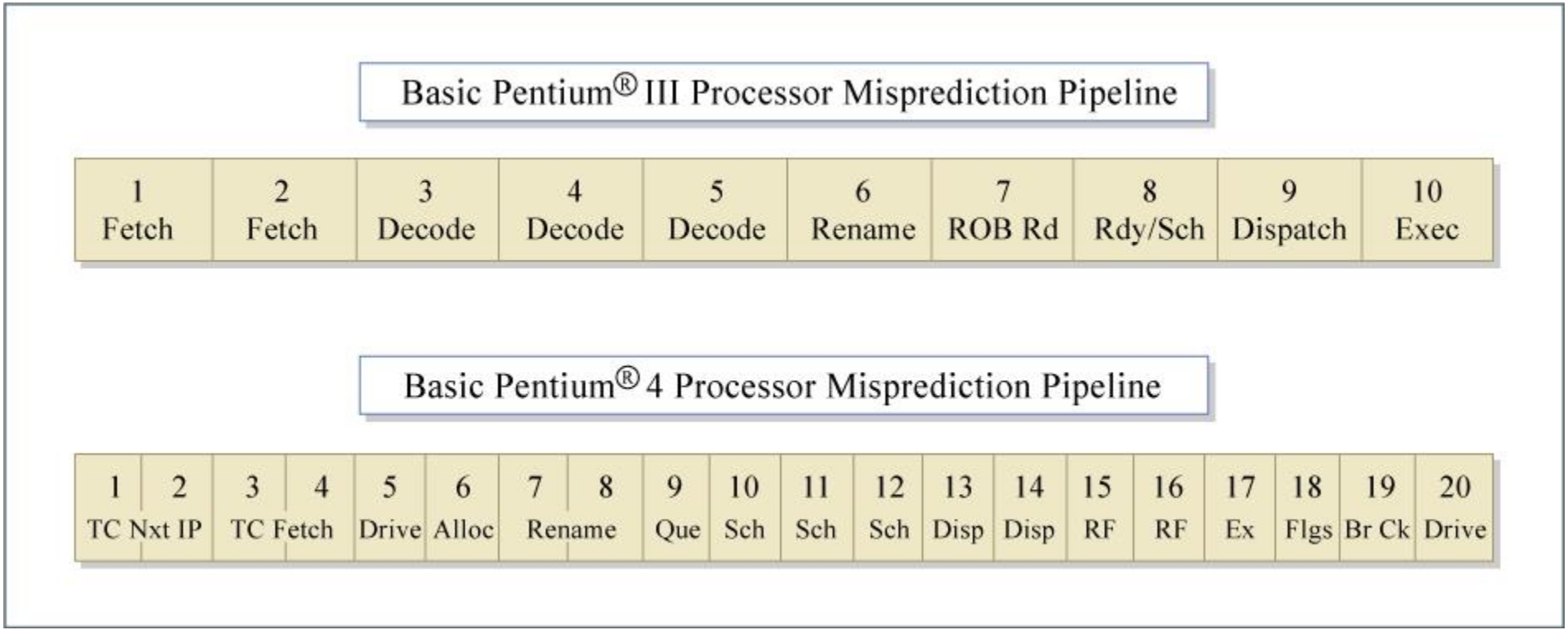
- W praktyce dla wielu programów predyktor bimodalny osiąga dokładność na poziomie 85-90%
- Czy taka dokładność jest wystarczająco wysoka?
- Jak poważny jest problem przewidywania skoków
  (jak bardzo kosztowne są pomyłki)?

Skala problemu przewidywania skoków

- 15-25% wszystkich instrukcji to skoki warunkowe (SPECint2006 - 17%)
- Niech N = odstęp (w cyklach) pomiędzy pobraniem instrukcji skoku a obliczeniem prawdziwości warunku skoku (N = *branch resolution latency*)
- W architekturach superskalarnych w 1 cyklu wysyłamy jednocześnie W instrukcji (W = *issue width*)
- Przy pomyłce predyktora tracimy N × W slotów instrukcji (*instruction slots*)

- Weźmy procesor superskalarny o parametrach N = 20, W = 5
- Ile cykli zajmie nam wykonanie 500 instrukcji?
  - Załóżmy, że 1 na 5 instrukcji to skok warunkowy
  - Dokładność 100%:
    - 100 cykli (wszystkie instrukcje pobrane z poprawnej ścieżki)
    - Brak zmarnowanej pracy
  - Dokładność 99%
    - 100 (poprawna ścieżka) + 20 (zła ścieżka) = 120 cykli
    - 20% instrukcji pobranych dodatkowo
  - Dokładność 98%
    - 100 (poprawna ścieżka) + 20 * 2 (zła ścieżka) = 140 cykli
    - 40% instrukcji pobranych dodatkowo
  - Dokładność 95%
    - 100 (poprawna ścieżka) + 20 * 5 (zła ścieżka) = 200 cykli
    - 100% instrukcji pobranych dodatkowo

Potoki w starszych procesorach superskalarnych



Predyktory korelujące

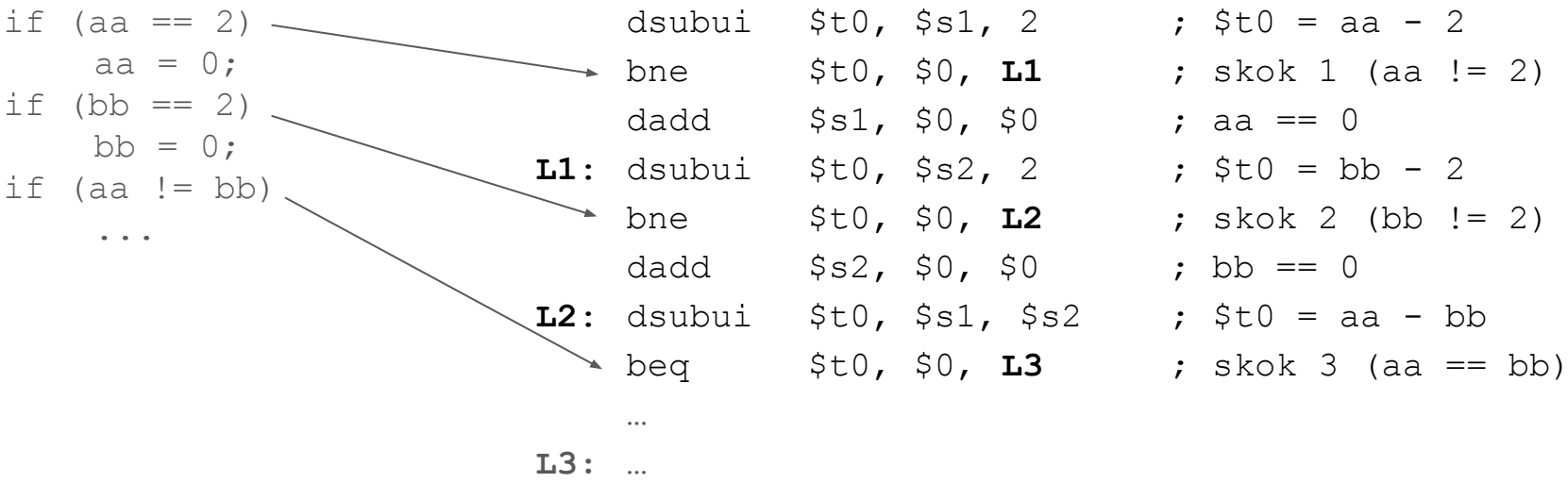
- Skoki warunkowe mogą wykazywać 2 rodzaje korelacji:
- **Korelacja globalna:** prawdziwość warunku skoku może zależeć od prawdziwości (lub nieprawdziwości) warunków w innych skokach wykonanych wcześniej
- **Korelacja lokalna:** prawdziwość warunku może zależeć od prawdziwości warunku we wcześniejszych wykonaniach tego samego skoku

Przykład korelacji lokalnej

```
for (int i = 0; i < 100; ++i)
    for (int j = 0; j < 3; ++j)
        ...
```

- Skok sprawdzający warunek kontynuacji pętli wewnętrznej wykazuje zachowanie “1110111011101110…”
- Można przewidzieć kierunek następnego skoku na podstawie 3 ostatnich wyników tego samego skoku

Przykład korelacji globalnej: `eqntott`, SPEC 1992



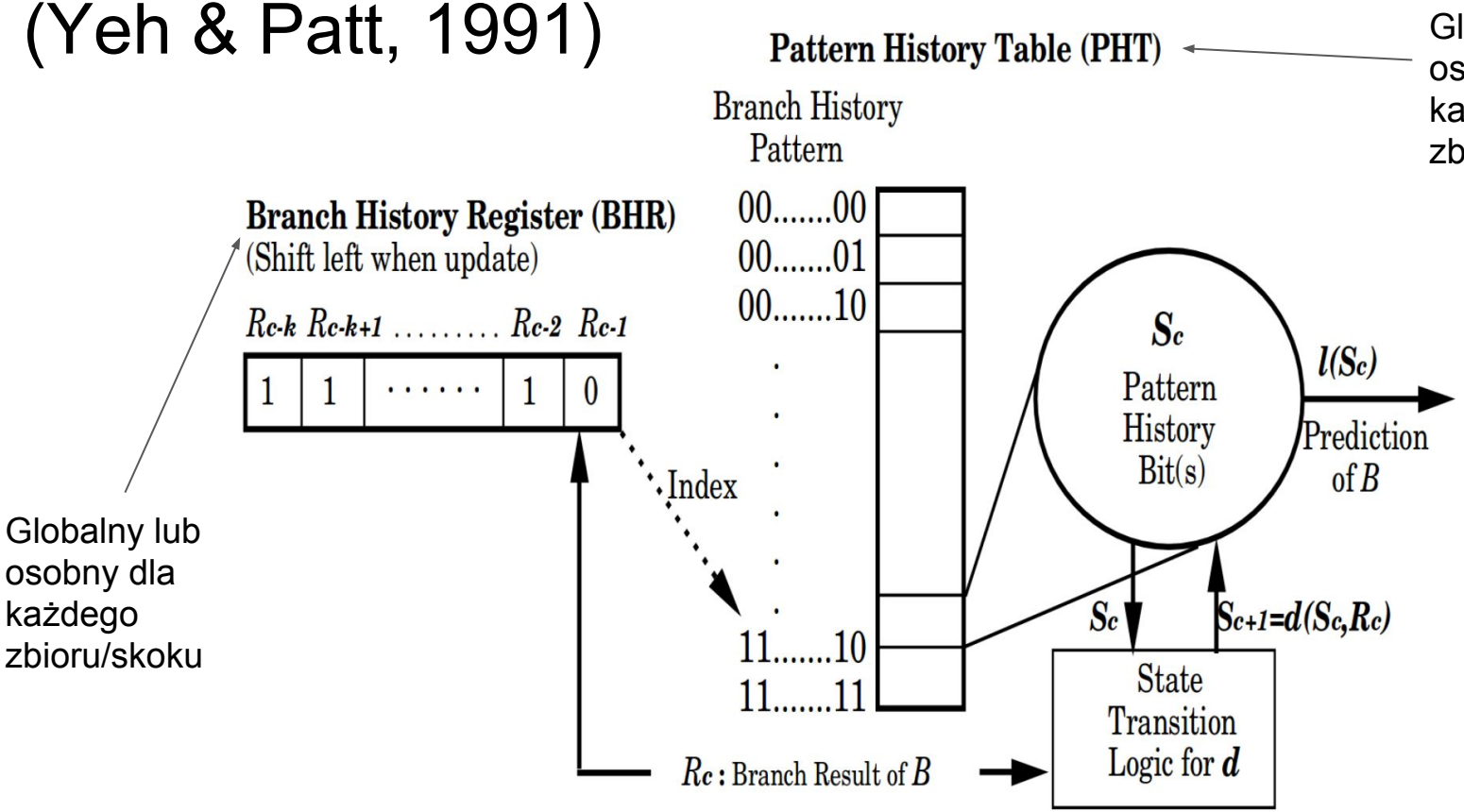
```
if (aa == 2)                dsubui   $t0, $s1, 2       ; $t0 = aa - 2
    aa = 0;                 bne      $t0, $0, L1        ; skok 1 (aa != 2)
if (bb == 2)                dadd     $s1, $0, $0        ; aa == 0
    bb = 0;             L1: dsubui   $t0, $s2, 2        ; $t0 = bb - 2
if (aa != bb)               bne      $t0, $0, L2        ; skok 2 (bb != 2)
    ...                     dadd     $s2, $0, $0        ; bb == 0
                        L2: dsubui   $t0, $s1, $s2      ; $t0 = aa - bb
                            beq      $t0, $0, L3        ; skok 3 (aa == bb)
                            …
                        L3: …
```

- Jeśli nie wykonały się skoki 1 i 2 (czyli `aa == 2` oraz `bb == 2`), to skok 3 na pewno się wykona

Dwupoziomowe predyktory adaptacyjne
(*Two-Level Adaptive Training Branch Prediction,* Yeh & Patt 1991)

- Historię ostatnich m skoków trzymamy w rejestrze historii skoków (BHR - *Branch History Register*). Istnieją trzy możliwości:
  - Jeden globalny BHR, w którym trzymana jest historia wszystkich skoków w programie (GA - *global adaptive*)
  - Osobny BHR dla każdej instrukcji skoku, w którym trzymana jest historia tylko tej instrukcji (PA - *per-address adaptive*)
  - Osobny BHR dla każdego zbioru (skoki dzielimy na zbiory np. patrząc na kilka bitów adresu) (SA - *set adaptive*)
- Historii używamy do indeksowania tablicy historii wzorców (PHT - *Pattern History Table*) zawierającej np. 2-bitowe liczniki
  - Tutaj też trzy możliwości: PHT globalne (g - *global*) lub osobne dla każdej instrukcji skoku (p - *per-address*) lub dla każdego zbioru (s - *set*)
- 9 możliwości: GAg, GAs, GAp, SAg, SAs, SAp, PAg, PAs, PAp

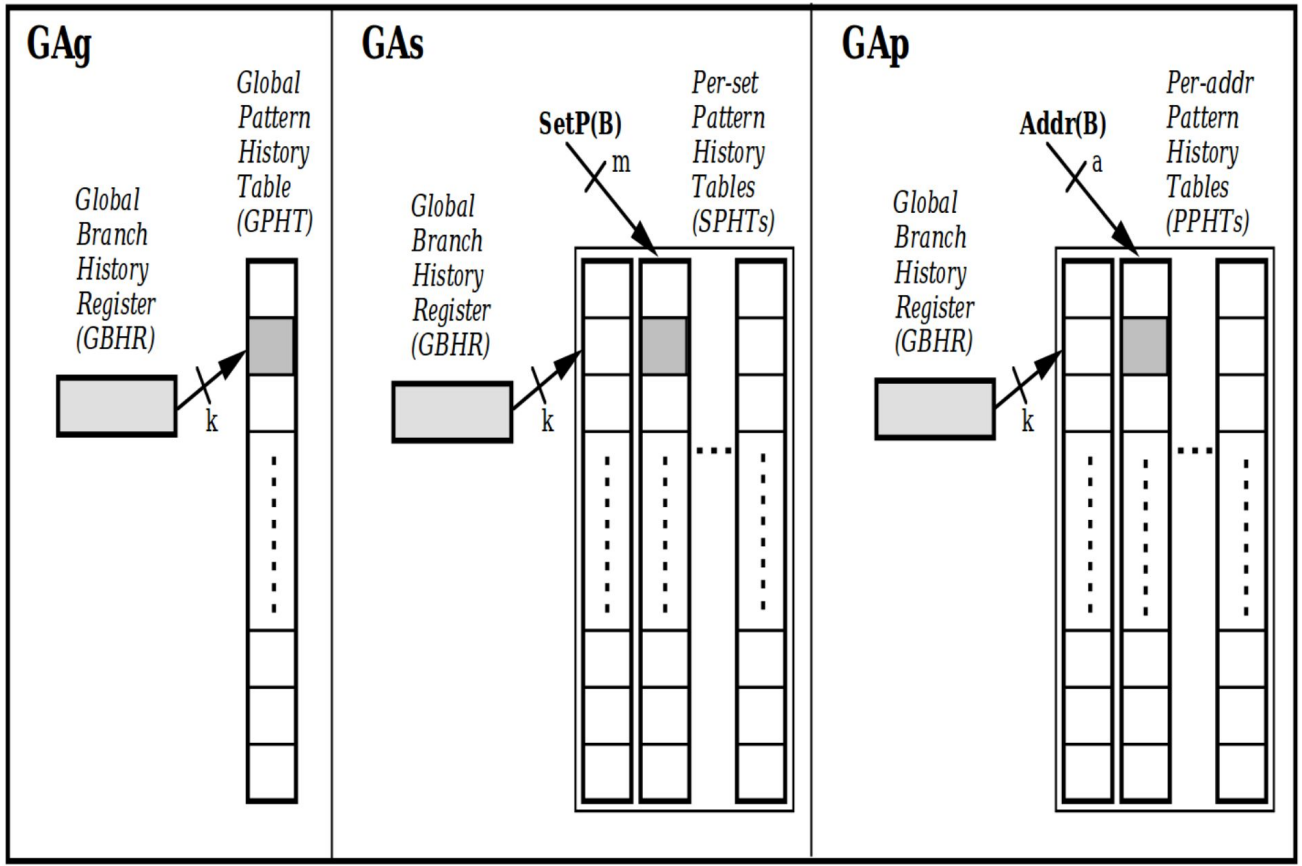
Dwupoziomowe predyktory adaptacyjne: schemat (Yeh & Patt, 1991)



Predyktory GAg, GAs i GAp

Wymagania pamięciowe (w bitach):

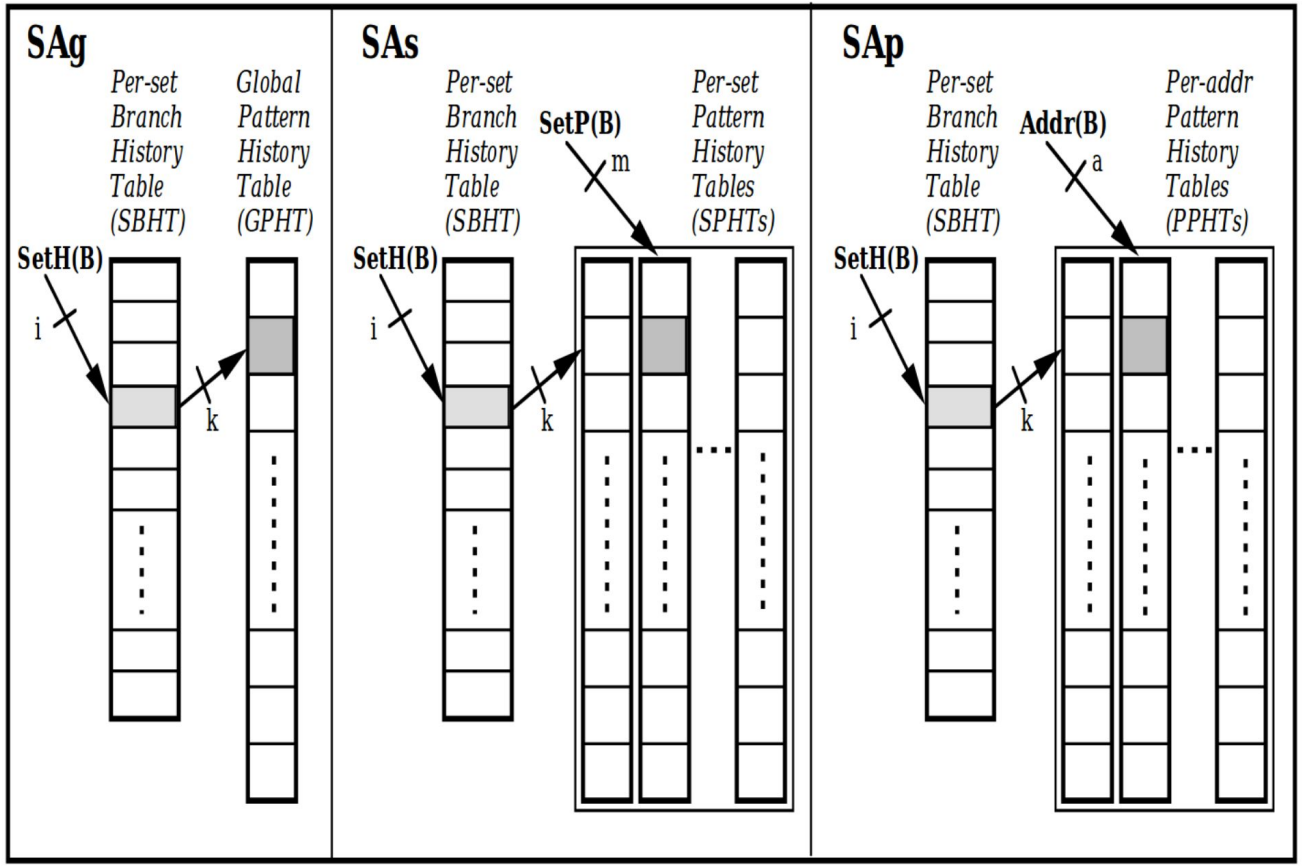
- GAg:
  $k + 2 \times 2^k$
- GAs:
  $k + 2 \times 2^k \times 2^m$
- GAp:
  $k + 2 \times 2^k \times 2^a$



Predyktory SAg, SAs i SAp

Wymagania pamięciowe (w bitach):

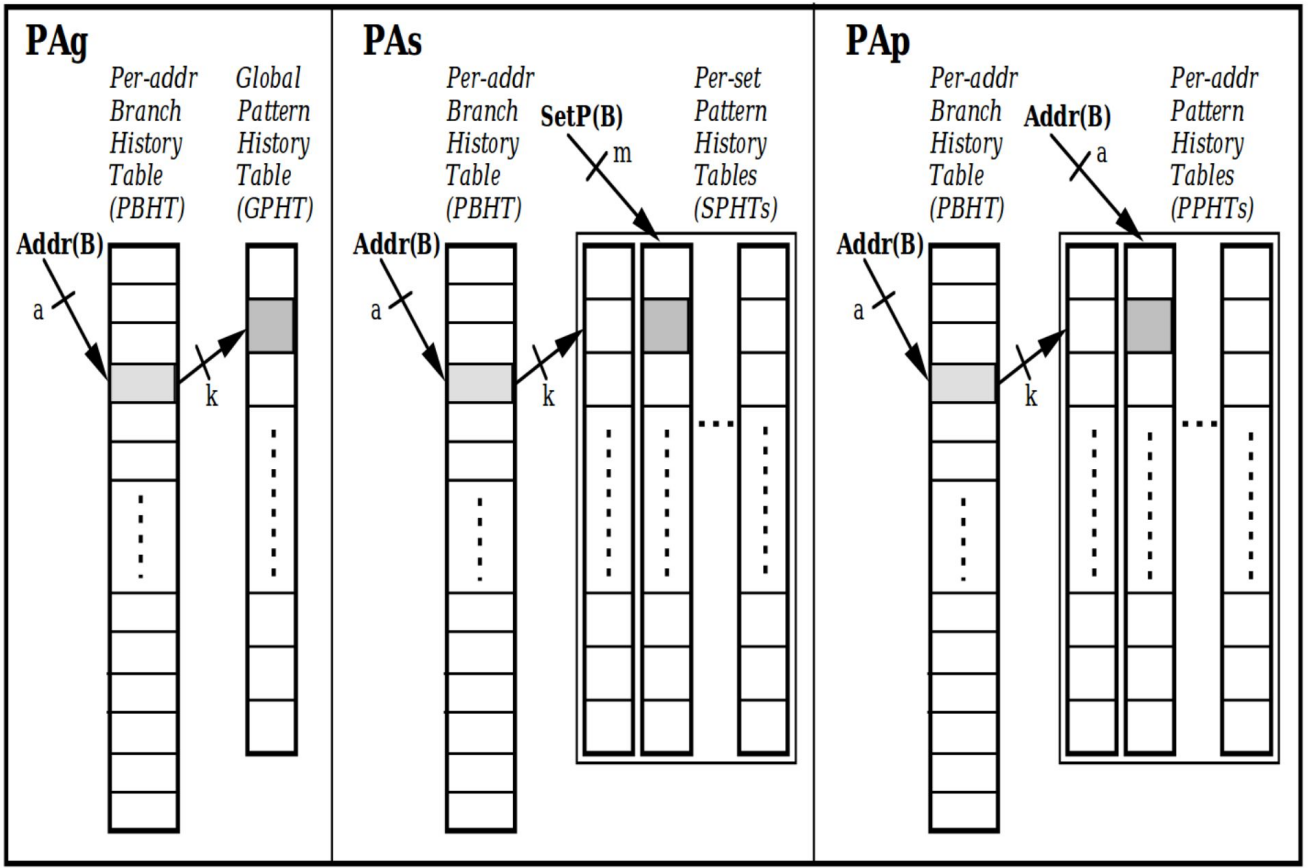
- SAg:
  $k \times 2^i + 2 \times 2^k$
- SAs:
  $k \times 2^i + 2 \times 2^k \times 2^m$
- SAp:
  $k \times 2^i + 2 \times 2^k \times 2^a$



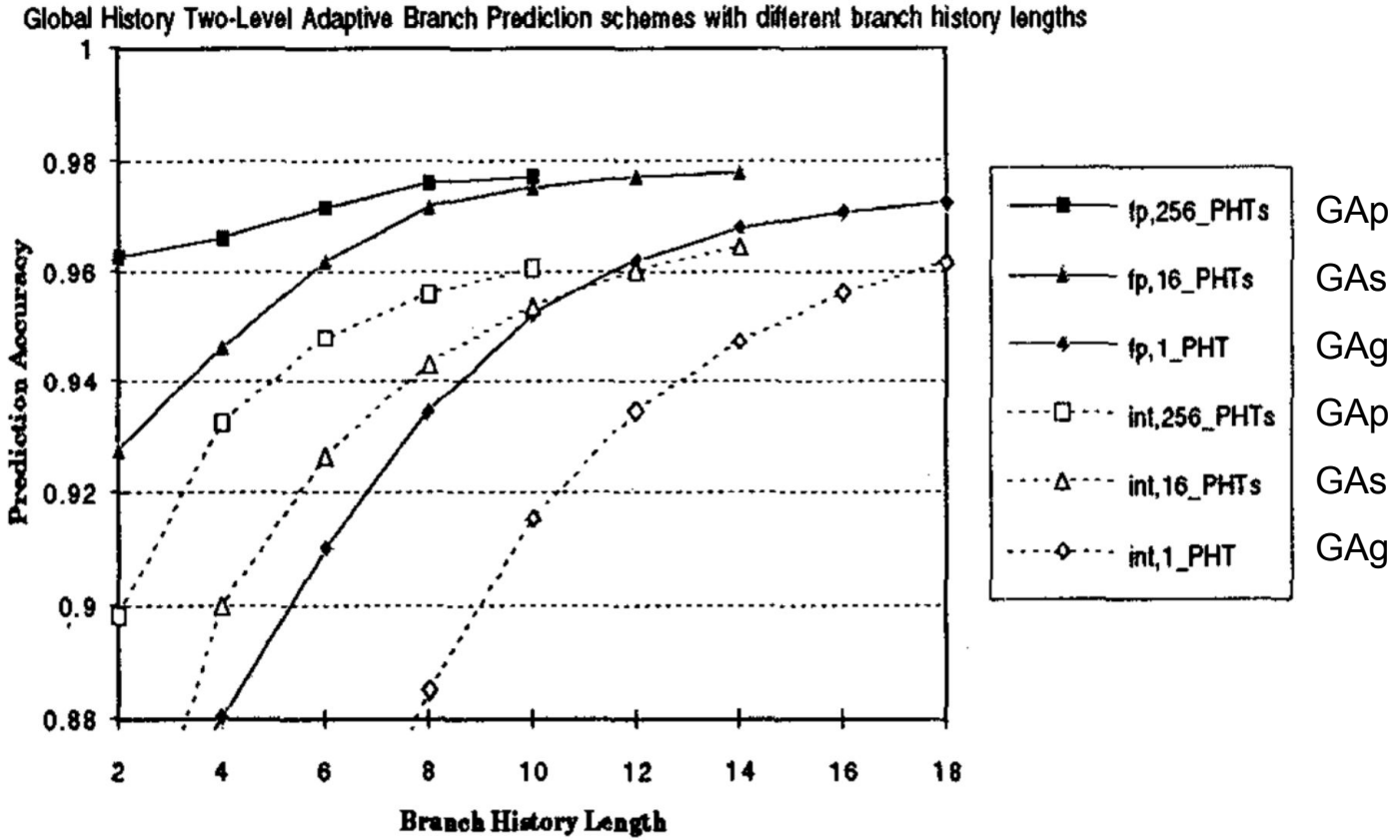
Predyktory PAg, PAs i PAp

Wymagania pamięciowe (w bitach):

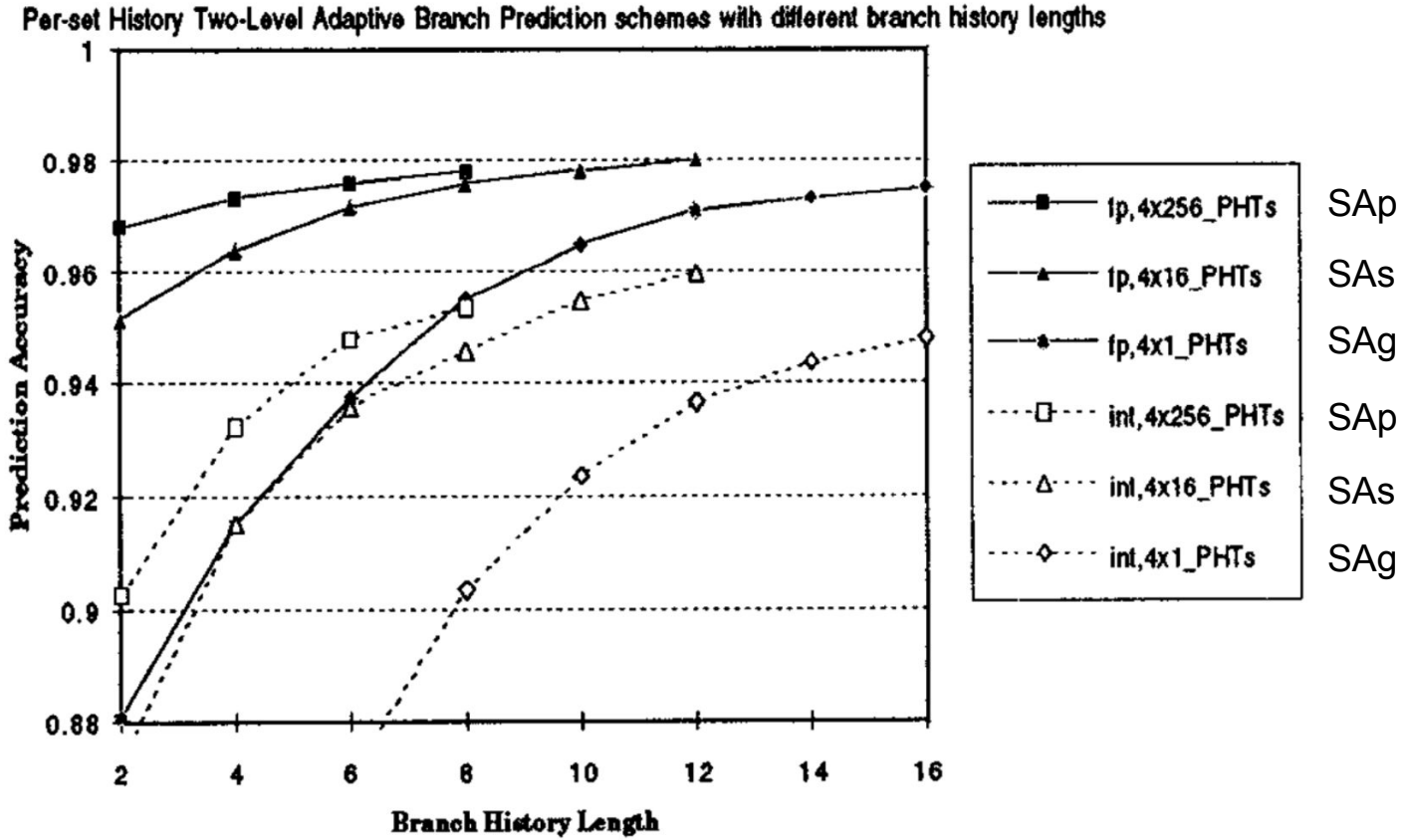
- PAg:
  $k \times 2^a + 2 \times 2^k$
- PAs:
  $k \times 2^a + 2 \times 2^k \times 2^m$
- PAp:
  $k \times 2^a + 2 \times 2^k \times 2^a$



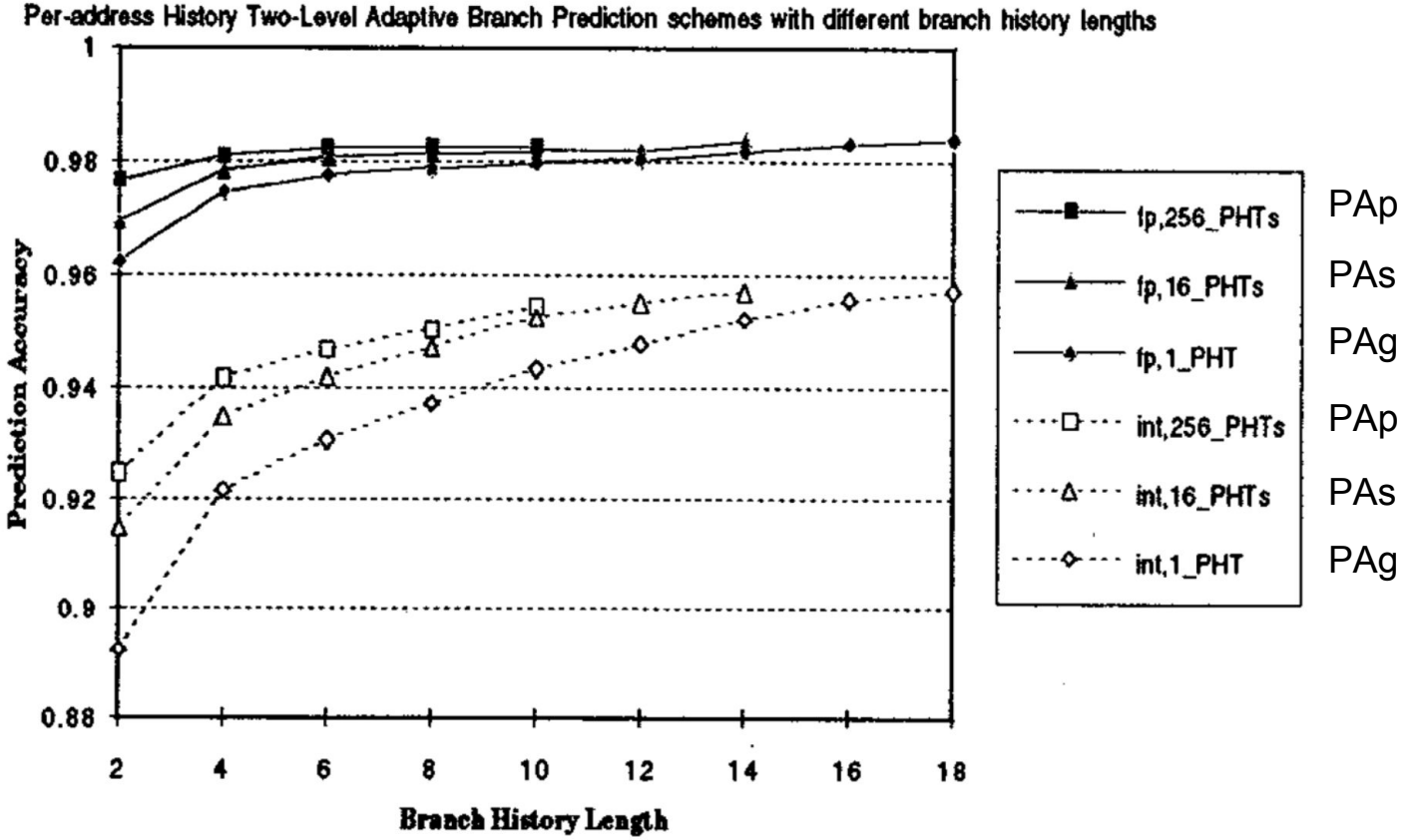
Dokładność: GAg, GAs i GAp (SPEC89)



Dokładność: SAg, SAs i SAp (SPEC89)



Dokładność: PAg, PAs i PAp (SPEC89)



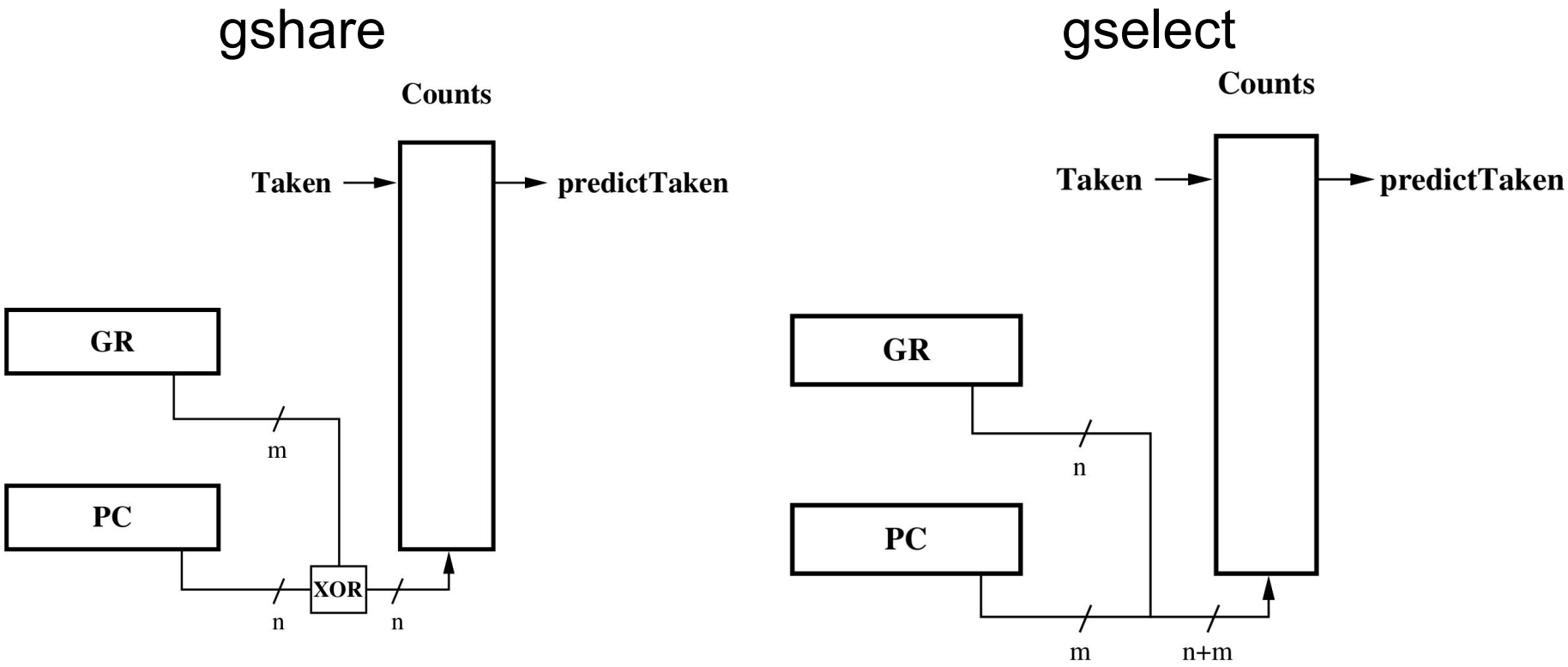
Różnice pomiędzy wariantami

- Predyktory GA* wykorzystują korelację pomiędzy różnymi skokami
- Predyktory PA* wykorzystują korelację w obrębie jednej instrukcji skoku
- Predyktory SA* robią coś pomiędzy (korelacja w obrębie jednego zbioru)

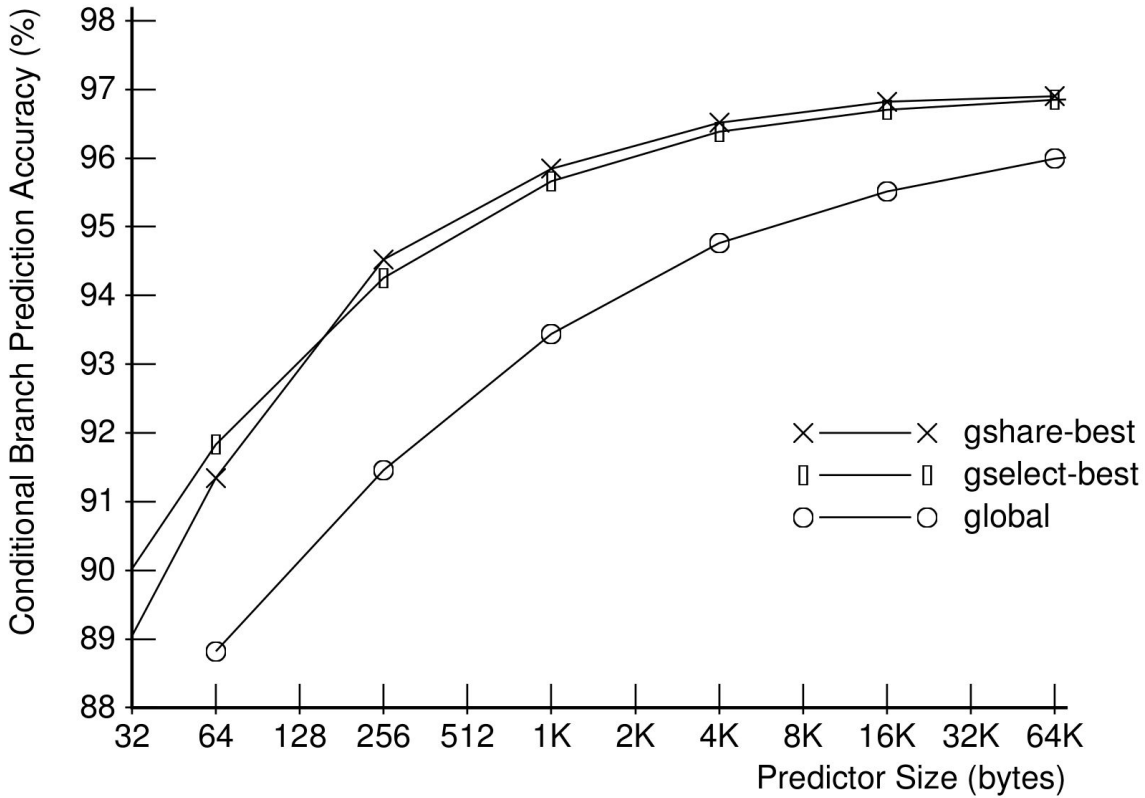
- W predyktorach *Ag występuje problem aliasingu (wielu instrukcjom skoku odpowiada jeden licznik w PHT)
- Predyktory *As ograniczają ten problem (jeden PHT na zbiór)
- Predyktory *Ap go eliminują (teoretycznie)
    - W praktyce niemożliwe jest mieć osobne PHT dla każdej instrukcji skoku

- Predyktory GA* nadają się do wykrywania wzorców w skokach w ciągłych, pozbawionych pętli, fragmentach kodu…
- … ale GAg cierpi z powodu aliasingu, a GAp i GAs nie są wydajne pamięciowio
- Czy jest jakiś tańszy sposób na uniknięcie aliasingu?
- gselect: GAg, ale do indeksowania PHT używamy konkatenacji kilku bitów adresu i historii
- gshare: GAg, ale do indeksowania PHT używamy XOR-a części adresu z historią

gshare i gselect: schemat



Porównanie GAg, gshare i gselect (SPEC89)



Predyktory hybrydowe
(*Combining Branch Predictors*, McFarling 1993)

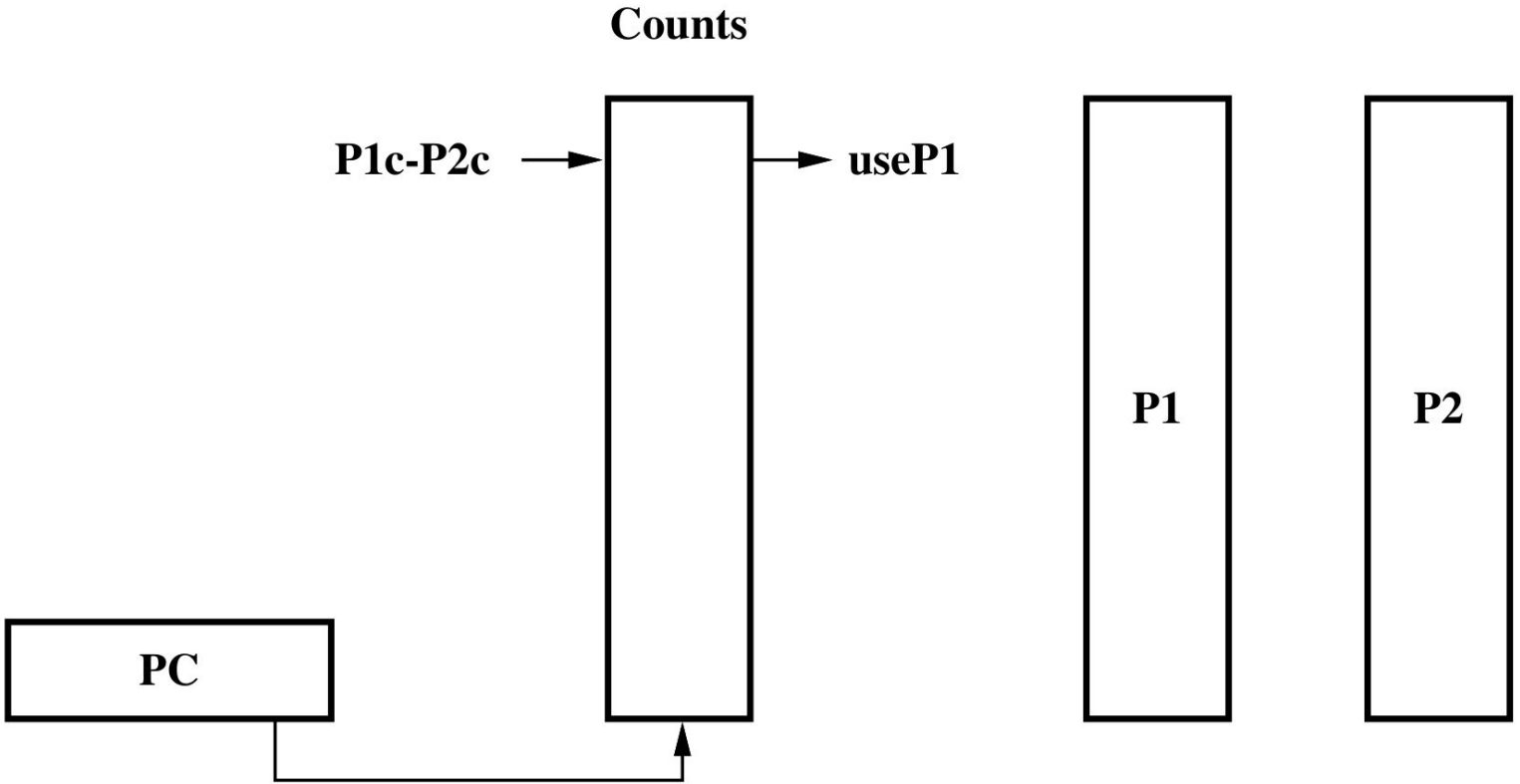
- Nie ma jednego doskonałego predyktora
  - Predyktory globalne nie wykryją korelacji lokalnej
  - Predyktory lokalne nie wykryją korelacji globalnej
  - Bardziej skomplikowane predyktory, w tym predyktory dwupoziomowe, wymagają czasu żeby tabele zapełniły się sensownymi informacjami (*warm-up time*)
- Pomysł: połączmy kilka predyktorów w jeden!
  - Np. predyktor bimodalny (krótki *warm-up time*, lokalny licznik dla każdego skoku) z gshare (dłuższy *warm-up time*, wykrywa korelacje pomiędzy skokami)
  - Przy każdym skoku trzeba się na jeden zdecydować
  - Predyktor hybrydowy jest tak dobry jak nasza umiejętność wyboru odpowiedniego podpredyktora

Wybór pomiędzy dwoma podpredyktorami

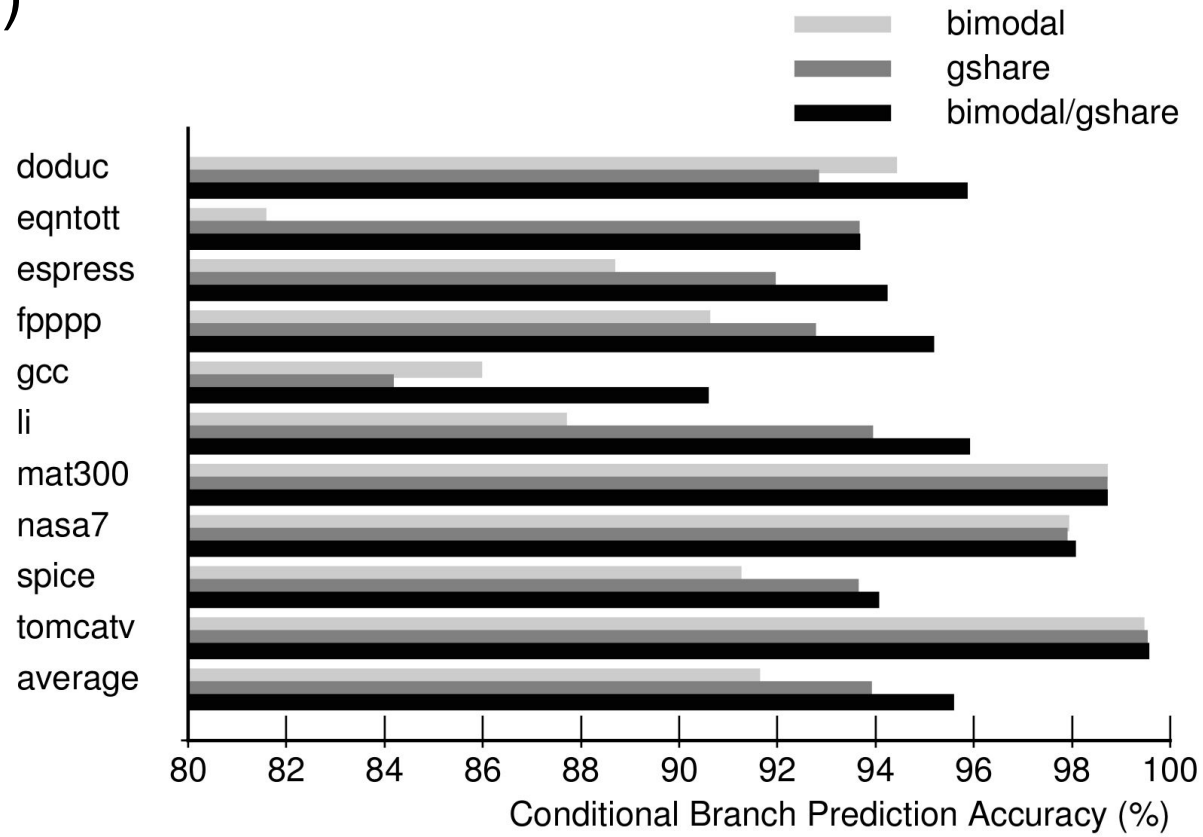
- Dla każdego skoku trzymamy informację o tym, który podpredyktor lepiej przewidywał kierunek tego konkretnego skoku (np. 2-bitowy licznik)
- Najbardziej znaczący bit = 1 → wybieramy podpredyktor 1, w p.p. podpredyktor 2
- Jeśli jeden z podpredyktorów przewidział kierunek poprawnie, a drugi nie, to odpowiednio aktualizujemy licznik

P1c	P2c	P1c-P2c	
0	0	0	(no change)
0	1	-1	(decrement counter)
1	0	1	(increment counter)
1	1	0	(no change)

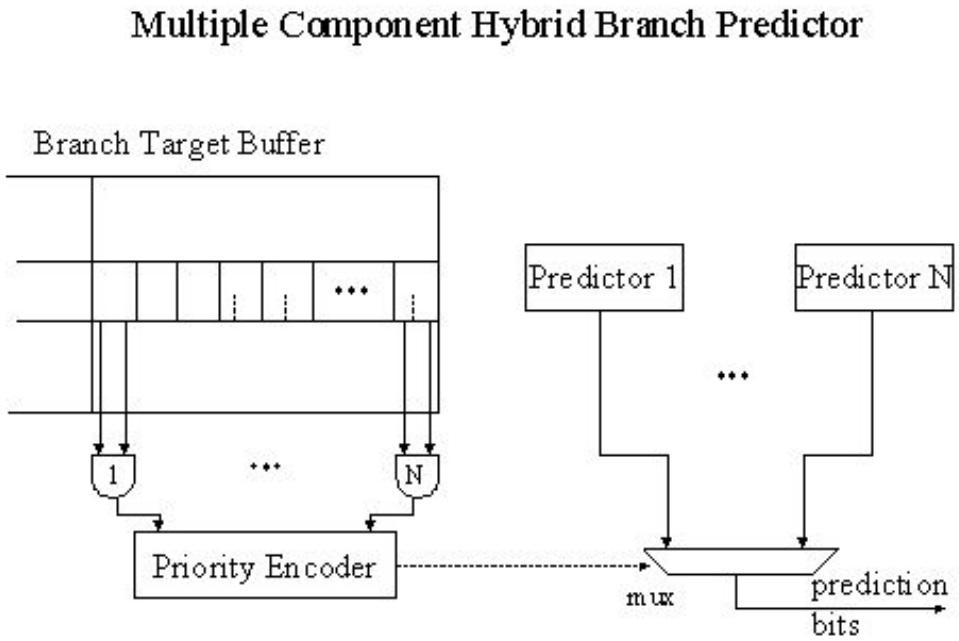
Predyktor hybrydowy z dwoma predyktorami: schemat



Predyktor hybrydowy z dwoma podpredyktorami: dokładność (SPEC89)



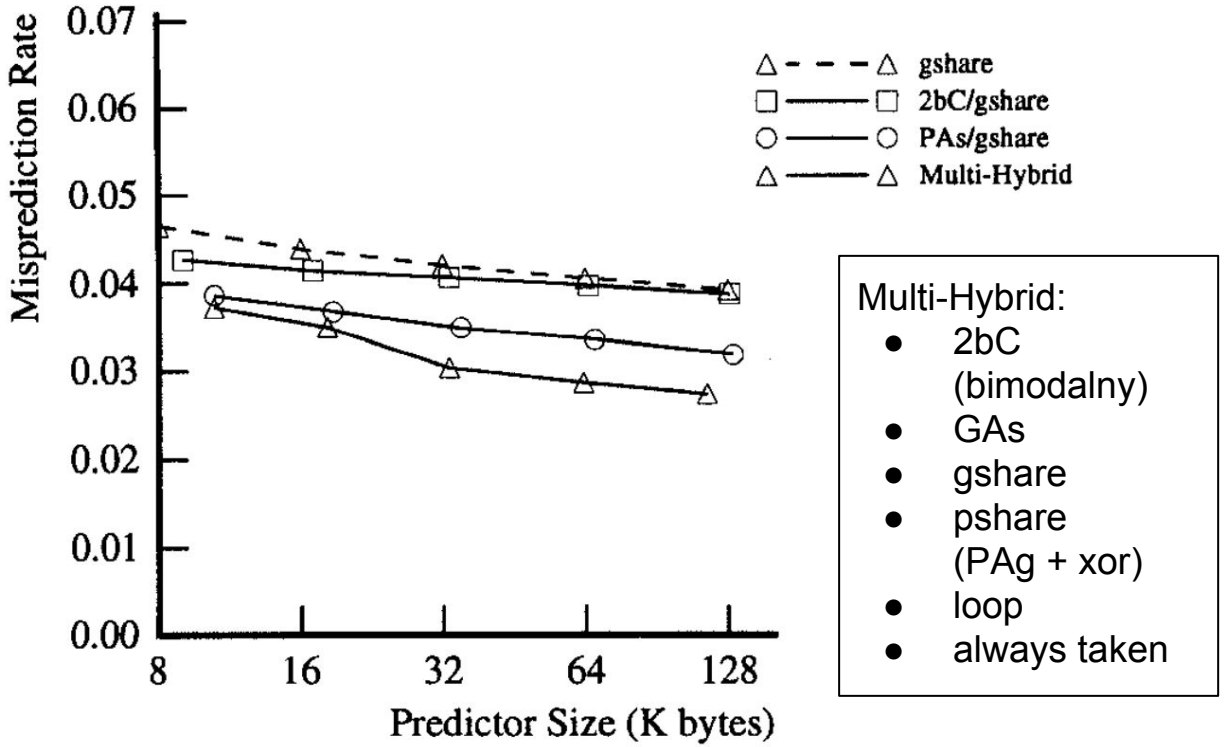
Uogólnienie: predyktor hybrydowy z N podpredyktorami (Evers et al., 1996)



Działanie uogólnionego predyktora hybrydowego

- Dla każdej instrukcji skoku trzymamy N 2-bitowych liczników
  - Na początku wszystkie mają wartość 3
- Wybieramy predyktor, którego licznik ma wartość 3
  - Zasada aktualizowania liczników gwarantuje że ≥1 licznik będzie miał wartość 3
  - Jeśli jest ich więcej to koder priorytetów (*priority encoder*) wybiera jeden na podstawie statycznie przydzielonych priorytetów
- Gdy poznamy wynik porównania:
  - Jeśli przynajmniej 1 predyktor którego licznik wynosił 3 się nie pomylił, to zmniejszamy licznik o 1 wszystkim predyktorom które się pomyliły
  - W p.p. zwiększamy licznik o 1 wszystkim predyktorom które się nie pomyliły

Dokładność uogólnionego predyktora hybrydowego (SPECint92)



Multi-Hybrid:

- 2bC (bimodalny)
- GAs
- gshare
- pshare (PAg + xor)
- loop
- always taken

Predyktor perceptronowy

(*Dynamic Branch Prediction with Perceptrons*, Jiménez & Lin 2003)

- Perceptron łączy n wejść $x_1, x_2, ..., x_n \in \{-1, 1\}$ w jedno wyjście $y \in \mathbb{Z}$
- Wewnętrzny stan perceptrona składa się z wag $w_0, w_1, \ldots, w_n$
- Perceptron oblicza y na podstawie równania $y = w_0 + \sum_{i=1}^{n} x_i w_i.$
- Skok wykonany jeśli $y \geq 0$, w p.p. niewykonany

Uczenie perceptronów

- Perceptron możemy nauczyć funkcji boolowskiej $f(x_1, x_2, \ldots, x_n)$ pod warunkiem, że jest ona liniowo separowalna
  - Dziedzinę funkcji f możemy podzielić hiperpłaszczyzną na część dla której f zwraca 0 i część dla której f zwraca 1
  - Np. XOR nie jest funkcją liniowo separowalną
- Niech t = 1 jeśli skok się wykonał, -1 w p.p.

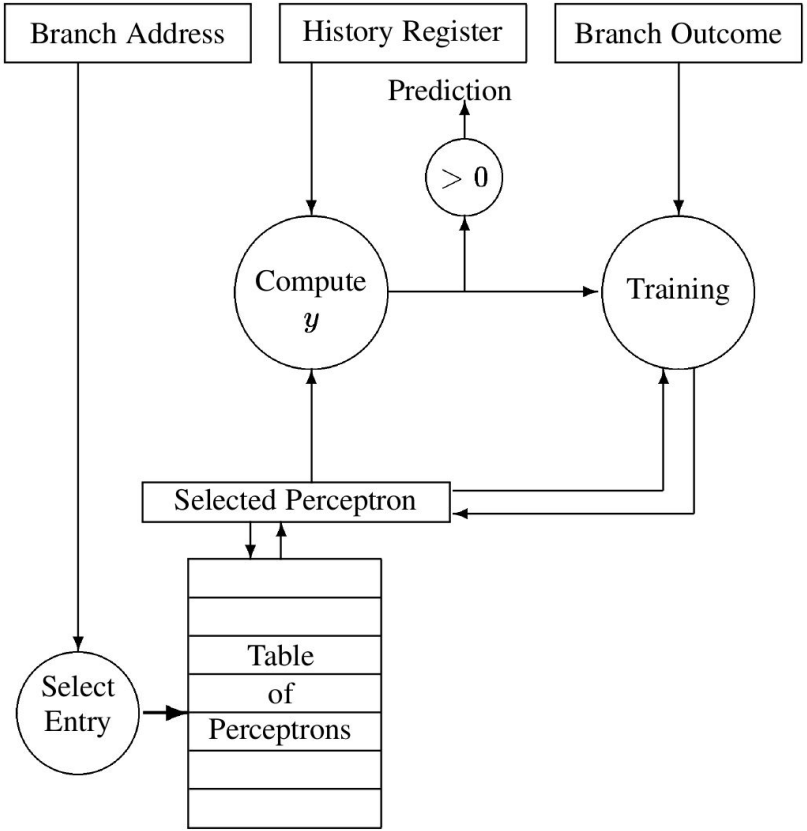
```
if sign(y) != t or |y| ≤ □  // □ - próg mówiący że wystarczy uczenia
    for i := 0 to n do
        w_i := w_i + t * x_i        // zawsze x_0 = 1
```

Zwiększamy wagę jeśli $t = x_i$, w p.p. zmniejszamy

Działanie predyktora perceptronowego

- Używamy adresu instrukcji skoku aby wybrać jeden perceptron z tablicy perceptronów
- Obliczamy wartość y korzystając z wag tego perceptronu i globalnej historii skoków
- Przewidujemy wykonanie skoku na podstawie y
- Gdy poznamy rzeczywisty wynik skoku, używamy algorytmu trenującego na wybranym perceptronie
- Nowe wagi zapisujemy w odpowiednim miejscu w tablicy perceptronów

Predyktor perceptronowy: schemat



Literatura

1. T.-Y. Yeh, Y. Patt, *“Two-level Adaptive Branch Prediction”, Proceedings of the 24th ACM/IEEE Int’l Symposium on Microarchitecture*, listopad 1991
2. T.-Y. Yeh, Y. Patt, *“Alternative Implementations of Two-level Adaptive Branch Prediction”, 19th Annual International Symposium on Computer Architecture,* Maj 1992
3. S. McFarling, *“Combining Branch Predictors”, WRL Technical Note TN-36,* Digital Equipment Corporation, czerwiec 1993
4. M. Evers, P.-Y. Chang, Y. Patt, *“Using Hybrid Branch Predictors to Improve Branch Prediction Accuracy in the Presence of Context Switches”, 23rd Annual International Symposium on Computer Architecture*, maj 1996
5. D. Jiménez, C. Lin, “*Dynamic Branch Prediction with Perceptrons”, Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, styczeń 2001