ILP – paralelizmus instrukční úrovně
(instruction level paralelism)

Osnova

- ILP
- Techniky používané kompilátory pro zvětšení míry ILP
- Rozbalení smyček
- Statická predikce skoků
- Dynamická predikce skoků
- Předcházení hazardům pomocí dynamického plánování
- Tomasulo algoritmus (začátek)
- Závěr

Přehled pipeliningu

- CPI (clock/cycles per instruction) pipeline = CPI ideální pipeline + vliv strukturních hazardů + vliv datových hazardů + vliv řídících hazardů
  - CPI ideální pipeline : Měřený maximální výkon, kterého je daná implementace v ideálním případě schopna dosáhnout
  - Strukturní hazardy: HW není schopen provádět současně danou kombinaci instrukcí (HW omezení)
  - Datové hazardy: Instrukce závisí na výsledku předchozí instrukce, která je stále v pipeline
  - Řídící hazardy: Způsobené zpožděním mezi načtením instrukce a rozhodnutím o změně instrukčního toku (větvení a skoky)

Paralelizmus instrukční úrovně

- Paralelizmus instrukční úrovně (ILP): znamená překrývání instrukcí za účelem zvýšení výkonu

- Dva přístupy k ILP:
  1. Závisí na hardware a využívá paralelizmus dynamicky (např. Pentium 4, AMD Opteron)
  2. Spočívá v softwarových technologiích, které vyhledávají paralelizmus staticky v době překladu (architektury VLIW, např. Itanium 2)

Paralelizmus instrukční úrovně (ILP)

- Základní blok programu (ZB) ILP je relativně malý
  - ZB: sekvence příkazů bez větvení kromě větvení na vstupu a na výstupu
  - Střední dynamická frekvence větvení 15% až 25%
    => mezi dvěma podmíněnými skoky se vykoná 4 až 7 instrukcí
  - Plus instrukce v ZB se zdají být závislé jedna na druhé
- Pro významné zvýšení výkonu musíme využít ILP přes více základních bloků
- Nejednodušeji: paralelizmus na úrovni smyček pro využití paralelismu mezi iteracemi. Např.:

```
for (i=1; i<=1000; i=i+1)
        x[i] = x[i] + y[i];
```

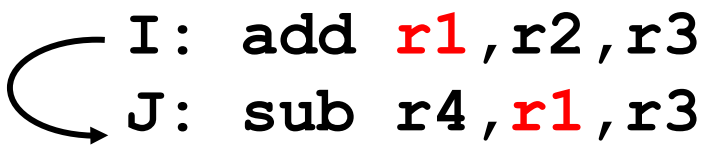
Paralelizmus úrovně smyček

- Využít úrovně paralelismu smyček jejich „rozbalením“ a to buď
  1. Dynamickou predikcí větvení nebo
  2. Staticky – rozbalení smyček kompilátorem

  (Jiný případ jsou vektory - bude zmíněno později)
- Určení závislostí instrukcí je kritické pro paralelizmus smyček
- Jestliže jsou 2 instrukce
  - paralelní, mohou být prováděny současně v pipeline libovolné hloubky bez toho, že by způsobily pozastavení (neuvažujeme strukturní hazardy)
  - závislé, nejsou paralelní a musejí být prováděny v pořadí, i když se někdy mohou překrýt

Závislosti dat a hazardy

- $Instr_J$ je datově závislá (pravá závislost) na $Instr_I$:
  1. $Instr_J$ se pokouší číst operand předtím, než jej $Instr_I$ zapíše



  2. Nebo $Instr_J$ je datově závislá na $Instr_K$, která je závislá na $Instr_I$
- Jsou-li dvě instrukce datově závislé, nemohou se vykonávat současně, ani se úplně překrýt
- Datová závislost v posloupnosti instrukcí
  $\Rightarrow$ datová závislost ve zdrojovém kódu $\Rightarrow$ originální datová závislost musí zůstat zachována
- Jestliže datová závislost způsobila hazard v pipeline, nazývá se Read After Write (**RAW**) hazard

ILP a datové závislosti, hazardy

- HW/SW musí zachovat posloupnost programu:
  Pořadí instrukcí bude zachováno, jestliže se budou provádět sekvenčně tak, jak je určeno zdrojovým programem
  - Závislosti jsou vlastností programů
- Přítomnost závislosti indikuje potenciální výskyt hazardu, ale aktuální hazard a délka pozastavení je vlastností pipeline
- Důležitost datových závislostí:

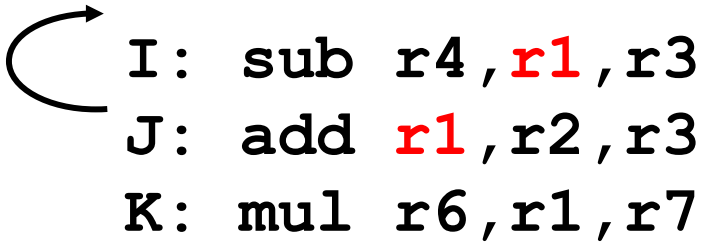
  1) indikují možnost výskytu hazardu

  2) určují pořadí, ve kterém se musejí počítat výsledky (mezivýsledky)

  3) udávají horní hranici možného paralelismu
- Cíl pro HW/SW: využít paralelizmus při zachování pořadí instrukcí jen tam, kde to přinese zisk

Závislost jmen #1: Anti-dependence

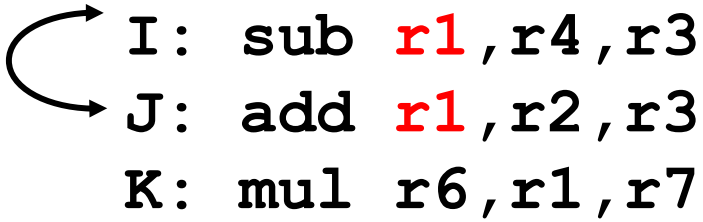
- Závislosti jmen: pokud 2 instrukce použijí stejný registr nebo místo v paměti označené jménem, ale mezi nimi není žádný tok dat spojený s tímto jménem; 2 verze závislosti jmen
- $Instr_J$ zapíše operand *předtím* než jej $Instr_I$ přečte



- Programátoři kompilátorů toto nazývají "anti-dependence". Pochází z opětovného použití jména "r1".
- Způsobí-li anti-dependence hazard v pipeline, nazývá se: Write After Read (**WAR**) hazard

Závislost jmen #2: Výstupní závislost

- $Instr_J$ zapíše operand *předtím*, než zápis provede instrukce $Instr_I$.



- Nazývá se "výstupní závislost" a pramení z opětovného použití jména "r1".
- Způsobí-li anti-dependence hazard v pipeline, nazývá se pak: Write After Write (**WAW**) hazard
- Instrukce figurující ve jmenné závislosti lze vykonat současně, pokud se změní jméno a instrukce pak nekolidují.
  - Přejmenování registrů řeší závislosti jmen pro registry
  - Řešení na úrovni kompilátoru nebo HW

Závislosti řízení výpočtu

- Každá instrukce je závislá (control dependent) na určitém souboru instrukcí větvení -. Obecně se musí tyto závislosti zachovat, aby byla zajištěna posloupnost programu.

```
if p1 {
    S1;
};
if p2 {
    S2;
}
```

- S1 je control dependent na p1 a S2 je control dependent na p2, ale nikoliv na p1.

Ignorování řídících závislostí

- Řídící závislost není nutné zachovat:
  - chceme-li provést instrukce, které by neměly být provedeny (porušily by se řídící závislosti), můžeme to dopustit, pokud korektnost programu zůstane zachována
- Naproti tomu, dvě vlastnosti jsou z hlediska korektního provedení programu kritické
  1) chování výjimek a
  2) tok dat

Chování výjimek

- Zachování ošetření výjimek
  $\Rightarrow$ libovolné změny v pořadí vykonávání instrukcí nesmí ovlivnit, jak jsou výjimky v programu ošetřeny
  ($\Rightarrow$ žádné nové výjimky)
- Příklad:

```
        DADDU    R2,R3,R4    (součet doubleword, unsigned)
        BEQZ     R2,L1       (větvení if EQ or Zero)
        LW     R1,0(R2)      (naplnění R1 z paměti)
L1:
```

  (Předpokládejme větvení nejsou opožděná)
- Můžeme přesunout LW před BEQZ?
- Pokud budeme ignorovat řídící závislost tak ano, ale load může způsobit výjímku při přístupu do paměti (memory protection exception)

Tok dat

- Tok dat: aktuální tok dat mezi instrukcemi, které produkují výsledky a těmi, které tyto výsledky používají. Větvení způsobuje, že tok je dynamický. Určete, která instrukce je zdrojem dat:

- Příklad:
```
        DADDU   R1,R2,R3
        BEQZ    R4,skip     ;větvení
        DSUBU   R1,R5,R6    ;můžu přesunout před větvení?
skip:   OR      R7,R1,R8
```

- OR závisí na DADDU nebo na DSUBU → ne
  Tok dat při provádění musí být zachován.

- Příklad2:
```
        DADDU R1, R2, R3
        BEQZ R12, skip      ;větvení
        DSUBU R4, R5, R6    ;můžu přesunout před větvení?
        DADDU R5, R4, R9
skip:   OR R7, R8, R9
```

- Můžu přesunout DSUBU před skok?
  Ano (za přepokladu že není R4 úžíváno za smyčkou) → porušení řídící závislosti nemusí znamenat chybné provedení

Osnova

- ILP
- Techniky používané kompilátory pro zvětšení míry ILP
- Rozbalení smyček
- Statická predikce skoků
- Dynamická predikce skoků
- Předcházení datovým hazardům dynamickým plánováním
- (Start) Tomasulo algoritmus
- Závěr

Softwarové techniky - příklad

- Tento program připočítává skalár s k vektoru X:

```
for (i=1000; i>0; i=i-1)
   x[i] = x[i] + s;
```

- Předpokládejme následující latence pro všechny příklady:
  - V těchto příkladech budeme **ignorovat** opožděná větvení

Instrukce dávající výsledek	*Instrukce používající výsledek*	*Latence [#cyklů]*	*Pozastavení [#cyklů]*
FP ALU op	**Another FP ALU op**	**4**	**3**
FP ALU op	**Store double**	**3**	**2**
Load double	**FP ALU op**	**1**	**1**
Load double	**Store double**	**1**	**0**
Integer op	**Integer op**	**1**	**0**

FP smyčka - kde jsou hazardy?

- Nejprve přeložíme do kódu MIPS:
  - Pro zjednodušení podmínky předpokládejme, že 8 je nejnižší adresa

```
      LUI R1,8008     ;R1= start address, i=1000
Loop: L.D F0,0(R1)    ;F0=vector element from (R1) = x[i]
      ADD.D   F4,F0,F2    ;add scalar from F2: F4=F0+F2 = x[i]+s
      S.D 0(R1),F4    ;store result: x[i]=F4(=x[i]+s)
      DADDUI  R1,R1,-8    ;decrement pointer 8B(DW) i--
      BNEZ    R1,Loop ;větvení R1!=zero: i!=0?
```

FP smyčka vykazující pozastavení

```
1 Loop:  L.D F0,0(R1)    ;F0=vector element
2     stall (protože F0, T=2)
3     ADD.D   F4,F0,F2    ;add scalar in F2
4     stall (protože F4, T=3)
5     stall
6     S.D 0(R1),F4    ;store result
7     DADDUI  R1,R1,-8    ;decrement pointer 8B (DW)
8     stall (protože R1, T=2)   ;assumes can't forward to branch
9     BNEZ    R1,Loop ;branch R1!=zero
```

Instrukce produkující výsledek	*Instrukce používající výsledek*	*Latence v hodinových cyklech*
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1

- 9 hodinových cyklů: Přepsat kód kvůli minimalizaci prodlev?

Upravená FP smyčka minimalizující prodlevy

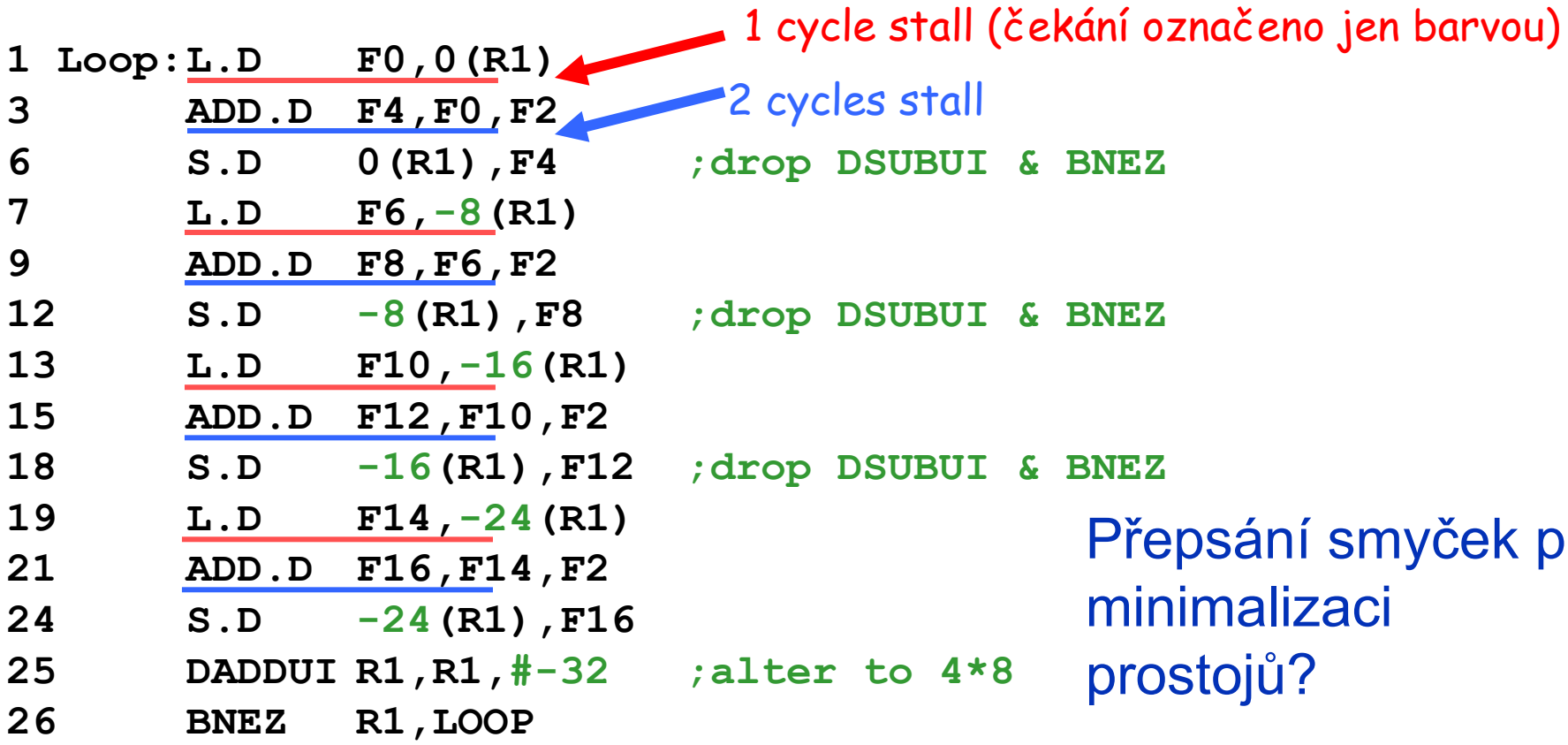
```
1 Loop:  L.D F0,0(R1)
2     DADDUI  R1,R1,-8      ;přesunuto z konce smyčky
3     ADD.D   F4,F0,F2      ;F0 už neblokuje
4     stall (protože F4)
5     stall
6     S.D 8(R1),F4 ;adresace (R1+8) kvulu presunu DADDUI
7     BNEZ R1,Loop
```

Zaměnit DADDUI a S.D

Instrukce dávající výsledek	*Instrukce používající výsledek*	*Latence v hodinových cyklech*
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1

7 hodinových cyklů, ale jen 3 pro výpočet (L.D, ADD.D,S.D), 4 pro režii smyčky.
Lze tento případ dále urychlit?

Čtyřnásobné rozvinutí smyčky



26 hodinových cyklů, nebo-li 6.5 cyklu na iteraci

(Předpokládejme, že R1 je násobkem 4)

Detaily rozbalování smyček

- Horní hranice smyčky není obvykle známa
- Předpokládejme, že to je n a chceme vytvořit k kopií těla smyčky
- Místo jednoduché rozvinuté smyčky generujeme dvojici po sobě jdoucích smyček:
  - Prvá se provede (n mod k) krát a její tělo tvoří originální smyčka
  - Druhou tvoří rozbalené tělo (oněch k kopií) obklopené vnější smyčkou, která se provádí (n/k) krát
- Pro velká n se největší část doby výpočtu odehraje v rozbalené smyčce

Rozbalování minimalizující prostoje
(využití více registrů)

```
1 Loop:L.D    F0,0(R1)
2      L.D    F6,-8(R1)
3      L.D    F10,-16(R1)
4      L.D    F14,-24(R1)
5      ADD.D  F4,F0,F2
6      ADD.D  F8,F6,F2
7      ADD.D  F12,F10,F2
8      ADD.D  F16,F14,F2
9      S.D    0(R1),F4
10     S.D    -8(R1),F8
11     S.D    -16(R1),F12
12     DSUBUI R1,R1,#32
13     S.D    8(R1),F16 ; 8-32 = -24
14     BNEZ   R1,LOOP
```

14 hodinových cyklů, nebo-li 3.5 cyklu na iteraci

Pět rozhodnutí při rozbalování

- Vyžaduje porozumění tomu, jak jedna instrukce závisí na druhé a jak mohou být změněny nebo přeskupeny za daných závislostí:
  1. Určení vhodného rozbalení smyčky nalezením nezávislých částí
  2. Použití různých registrů, aby se snížila zbytečná omezení (použití stejných registrů pro různé výpočty)
  3. Odstranění zbytečných instrukcí větvení a stanovení kódu iterace a ukončení smyčky
  4. Operace Čtení a Zápis mohou být v rozbalených smyčkách zaměněny, pokud jsou tyto operace z rozdílných iterací nezávislé (iterace = průchod smyčkou)
     - Transformace vyžaduje analýzu paměťových referencí a nalezení těch, které se nevztahují ke stejné adrese
  5. Plánování kódu, s respektováním všech závislostí, které jsou třeba, abychom dostali stejný výsledek jako u originálního kódu.

Tři limity rozbalování smyček

1. Snižování velikosti režie je na druhé straně „kompenzováno“ s každým extra rozbalením
   - Amdahlův zákon (aneb klesající výnosy s výšším stupněm rozbalení)
2. Nárůst velikosti kódu
   - Velké smyčky mohou snižovat úspěšnost cache !!! (nárůst „miss rate“)
3. „Tlak“ na registry: potenciální nedostatek registrů vlivem agresivního rozbalování
   - Není-li možné umístit všechny „živé“ hodnoty do registrů, část výhod se ztratí

- Rozbalení smyček redukuje důraz na větvení v pipeline; jinou cestou je predikce větvení

Predikce skoků

Druhy predikce:

- Statická predikce skoků
- Dynamická predikce skoků

Predikce poskytuje informace:

- Vlastní predikce - skok se koná/nekoná (Taken/NotTaken)
- Často také adresu cíle skoku kvůli urychlení
- Někdy i několik instrukcí, ležících v cíli skoku. To poskytne dobu na načtení instrukcí z hlavní paměti, které velmi pravděpodobně nejsou v tomto okamžiku v instrukční cache

Predikce skoků

Typická úspěšnost predikce: 90% - 96% (!)

- 4% - 10% špatná predikce
- Typicky 10-25 (úspěšně predikovaných) skoků mezi špatnými predikcemi,

  50 – 125 instrukcí mezi špatnými predikcemi

Cena za špatnou predikci roste

- s hloubkou pipeline
- s „šířkou“ stroje (počet paralelně vkládaných instrukcí)

Predikce – dnes nutná podmínka pro dobrý výkon

Predikce

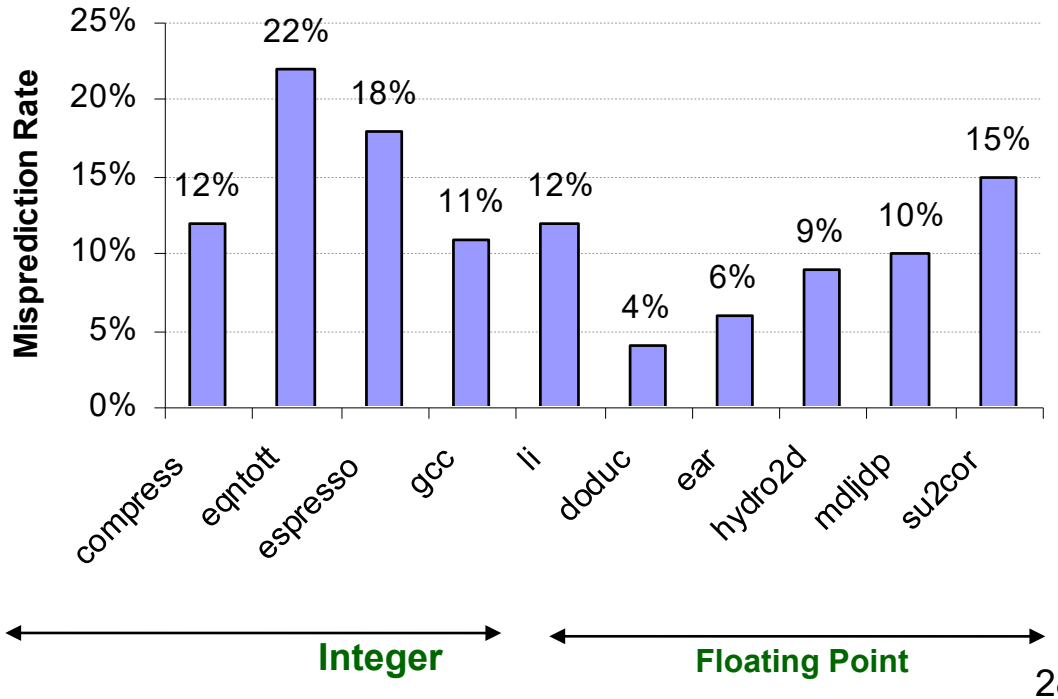
Proč predikce funguje?

- Algoritmy vykazují regularity
- Zpracovávaná data vykazují regularity
- Posloupnosti instrukcí obsahují redundance - zbytky toho, jak programátoři, popř. kompilátory nepřímo „promýšlejí“ problémy

Statická predikce skoků

- Přeskupení kódu kolem instrukcí větvení vyžaduje statickou predikci skoků v době kompilace
- Nejjednodušší predikční schéma je předpokládat, že se skok provede
  - Střední četnost špatné predikce = frekvence neprovedených skoků = 34% SPEC

Přesnější mechanismy predikují na základě informací získaných při minulých bězích programu a modifikují predikci založenou na posledním běhu:



Dynamická predikce skoků

- Je dynamická predikce skoků lepší než statická?
  - Zdá se, že tomu tak je
  - Programy obsahují menší počet důležitých větvení, která mají dynamické chování

Dynamická predikce

Branch-Prediction Buffer *(branch history table):*

- Nejjednodušší postup je odhadnout, zda se skok bude nebo nebude konat.
- Pomáhá v pipeline tam, kde zpoždění skoku je větší než čas, potřebný k určení možné cílové hodnoty PC.
  - *Můžeme-li ušetřit čas rozhodování, můžeme skákat dříve.*
- *Poznámka: Toto schéma nepomáhá u MIPSu, kterým jsme se zabývali.*
  - *Důvod: Rozhodnutí o skoku a cílový PC se počítá v I/D fázi za předpokladu, že nevznikl hazard mezi testovanými registry.*

Sedm schémat predikce

1. 1-bitový prediktor větvení
2. 2-bitový prediktor větvení
3. Prediktor s korelací
4. Kombinovaný prediktor větvení (Tournament Branch Predictor)
5. Cache pro cíle skoků (Branch Target Buffer)
6. Integrovaná jednotka pro předvýběr instrukcí
7. Prediktor návratové adresy

Řešení

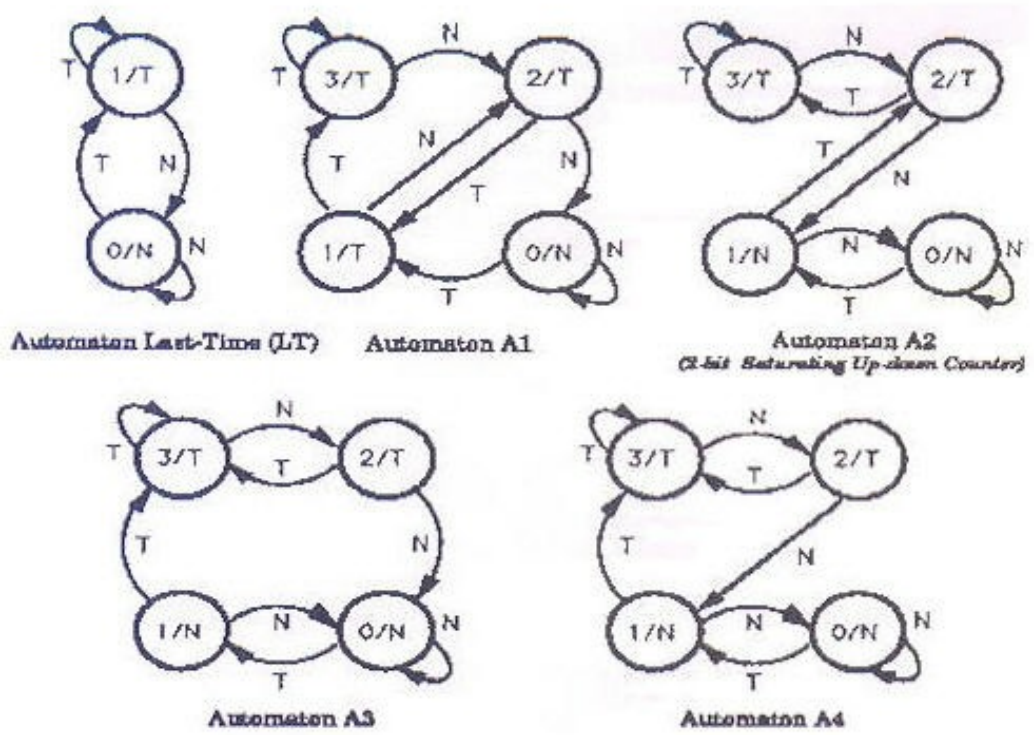
- Určit výsledek skoku co nejdříve, jak je možno
- Predikce skoku
  - Predikce podmínky skoku & cíle skoku
  - Předvýběr instrukcí z místa cíle skoku ještě před jeho vyhodnocením
  - Spekulativní výpočet
    - Instrukce z místa cíle skoku jsou přečteny a začnou se vykonávat ještě před vyhodnocením skokové instrukce
  - Jednoduché řešení
    - PC <- PC + 4, implicitní předvýběr další (sekvenční) instrukce
  - Při chybném odhadu se pipeline musí *zahodit*,
    - Příklad: chybná predikce je 10%, 4-jednotková 5-stupňová pipeline ztratí ~23% taktů!
      - Při 5% chybné predikci, se ztratí jen 13% taktů (time slot).
  - Potřebujeme přesnější predikci, abychom redukovali ztrátu vlivem chybného odhadu
    - Čím hlubší a širší je pipelinning, tím větší jsou ztráty, vzniklé chybnou predikcí!

Příklad ztrát při chybné predikci

```
1                LD      R1 <- A
2                LD      R2 <- B
3                MULT    R3, R1, R2
4                BEQ     R1, R2, TARGET
5                SUB     R3, R1, R4
6                ST      A <- R3
7    TARGET:     ADD     R4, R1, R2
```

Různé typy automatů

S(i): Stav v čase i
G(S(i)) -> T/F: Predikční rozhodovací funkce
E(S(i), T/N) -> S(i+1): Přechodová stavová funkce
Výkon (úspěšnost): **A2** (obvykle nejlepší), **A3**, **A4** následovány **A1** a nakonec **LT**
Pozn. T=taken(skok), N=not taken(neskok)
Stav = predikce, hrany/přechody = skutečnost



Počet bitů prediktoru

Benchmark	Prediction Accuracy (Overall CPI Overhead)			
	3-bit	2-bit	1-bit	0-bit
spice2g6	97.0 (0.009)	97.0 (0.009)	96.2 (0.013)	76.6 (0.031)
doduc	94.2 (0.003)	94.3 (0.003)	90.2 (0.004)	69.2 (0.022)
gcc	89.7 (0.025)	89.1 (0.026)	86.0 (0.033)	50.0 (0.128)
espresso	89.5 (0.045)	89.1 (0.047)	87.2 (0.054)	58.5 (0.176)
li	88.3 (0.042)	86.8 (0.048)	82.5 (0.063)	62.4 (0.142)
eqntott	89.3 (0.028)	87.2 (0.033)	82.9 (0.046)	78.4 (0.049)

- Velikost tabulky historie větvení: Přímo mapované pole s 2K položkami
- Programy (např. gcc) mohou mít více než 7000 instrukcí větvení
- Při kolizi sdílí různé instrukce větvení stejný prediktor (potenciálně chybná predikce).

Příklad

- Předpokládejme následující program, napsaný v C:

```
x = 1;
for (i = 1; i < 7; i ++) {
    x += x*i;
}
```

- Napište odpovídající program pro MIPS

Příklad

```
x = 1;
  for (i = 1; i < 7; i ++) {
    x += x*i;
  }
```

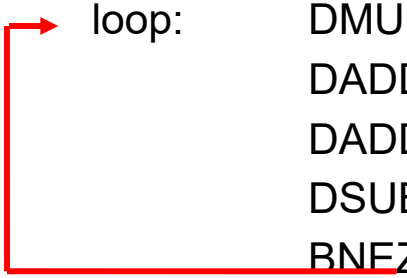
T … taken (skáče se)
NT … not taken (neskáče se)

```
        DADDI   R1, R0, #1    // x = 1
        DADDI   R2, R0, #1    // i = 1
loop:   DMUL    R3, R1, R2    // R3 = x * i
        DADD    R1, R1, R3    // x = x + R3
        DADDI   R2, R2, #1    // i = i + 1
        DSUBI   R4, R2, #7    // R4 = i - 7
        BNEZ    R4, loop      // if R4 != 0 goto loop
```

- Předpokládáme-li, že smyčka bude opakovaně prováděna, jaká bude posloupnost chování skoků? (T, NT)

Příklad

```
        DADDI   R1, R0, #1      // x = 1
        DADDI   R2, R0, #1      // i = 1
loop:   DMUL    R3, R1, R2      // R3 = x * i
        DADD    R1, R1, R3      // x = x + R3
        DADDI   R2, R2, #1      // i = i + 1
        DSUBI   R4, R2, #7      // R4 = i - 7
        BNEZ    R4, loop        // if R4 != 0 goto loop
```



- Předpokládáme-li, že smyčka bude opakovaně prováděna, jaká bude posloupnost chování skoků? (T, NT)

T, T, T, T, T, NT, T, T, T, T, T, NT, T, T, T, T, T, NT, . . .

Příklad

T, T, T, T, T, NT, T, T, T, T, T, NT, T, T, T, T, T, NT, . . .

- Předpokládejme 1-bitový prediktor větvení, inicializovaný na NT
- Smyčku budeme provádět 100 krát
- Jaký je podíl špatných předpovědí?

Příklad

T, T, T, T, T, NT, T, T, T, T, T, NT, T, T, T, T, T, NT, . . .

- Předpokládejme 1-bitový prediktor větvení, inicializovaný na NT
- Smyčku budeme provádět 100 krát
- Jaký je podíl špatných předpovědí?

1st branch: wrong
2nd branch: right
3rd branch: right
4th branch: right
5th branch: right
6th branch: wrong
7th branch: wrong
. . .

Jeden průchod smyčkou

100 provedení
Každé provedení: 4 správné, 2 špatné
četnost špatné predikce: 100 * 2 / 100 * 6
= 33%

(nezávisí na počtu běhů smyčky)

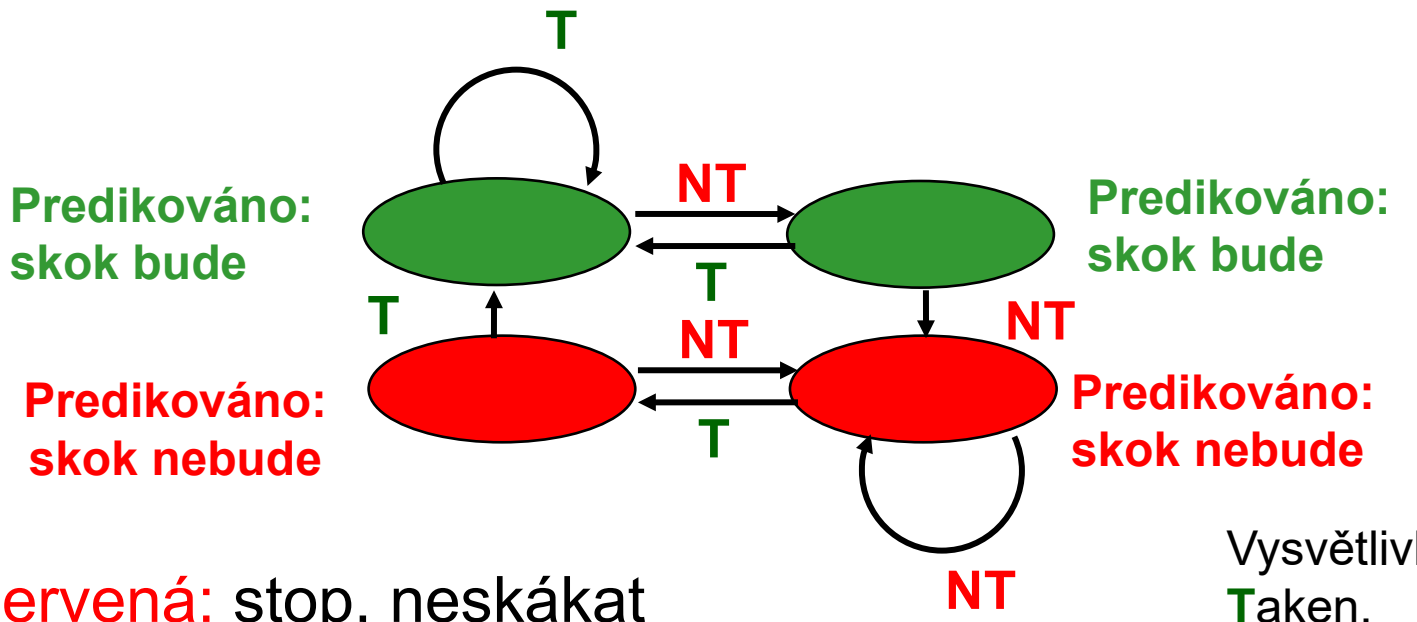
Příklad

T, T, T, T, T, NT, T, T, T, T, T, NT, T, T, T, T, T, NT, . . .

- Předpokládejme 2-bitový prediktor větvení, inicializovaný na NT, NT
- Smyčku budeme provádět 100 krát
- Jaký je podíl špatných předpovědí?

Dynamická predikce skoků

- Řešení: 2-bitové schéma, kde se predikce mění, dojde-li ke špatné predikci *dvakrát po sobě*



Vysvětlivka:
Taken,
Not **T**aken

- Červená: stop, neskákat
- Zelená: skok se bude konat
- Přidává *hysterezi do rozhodovacího procesu*

Příklad

T, T, T, T, T, NT, T, T, T, T, T, NT, T, T, T, T, T, NT, . . .

- Předpokládejme 2-bitový prediktor větvení, inicializovaný na NT, NT
- Smyčku budeme provádět 100 krát
- Jaký je podíl špatných předpovědí?

1st branch: wrong
2nd branch: wrong
3rd branch: right
4th branch: right
5th branch: right
6th branch: wrong
7th branch: right
8th branch: right
9th branch: right
10th branch: right
11th branch: right
12th branch: wrong
. . .

Prvé provedení: 3 špatné, 3 správné
Dalších 99 provedení: 1 špatná, 5 správné

podíl nezdařených předpovědí:
(3 + 99 * 1) / (6 * 100) ~ 17%

Bude dávat 3-bitový prediktor lepší výsledky?

Predikce větvení s korelací

- Dvoubitový prediktor používá pouze historii toho skoku, který právě predikuje
- Jednoduchá myšlenka: vzít do úvahy i ostatní skoky, protože chování skoků může být korelované
- Příklad z učebnice

```
if (d == 0)
    d = 1;
if (d == 1) {
    . . .
}
```

Příklad

```
if (d == 0)
    d = 1;
if (d == 1)
    . . .
}
```

Předpokládejme, že R1 obsahuje hodnotu d

```
        BNEZ    R1, L1      // b1
        DADDIU  R1, R0, #1
L1:     DSUBUI  R3, R1, #1
        BNEZ    R3, L2      // b2
        . . .
L2:     . . .
```

Provedení instrukcí větvení je závislé – zde závislost zprostředkovává proměnná d

Příklad

Předpokládejme, že R1 obsahuje hodnotu d

```
if (d == 0)
    d = 1;
if (d == 1)
    . . .
}
```

```
        BNEZ    R1, L1      // b1
        DADDIU  R1, R0, #1
L1:     DSUBUI  R3, R1, #1
        BNEZ    R3, L2      // b2
        . . .
L2:     . . .
```

Výchozí hodnota d	d == 0?	b1	Hodnota d před b2	d == 1?	b2
0	ano	NT	1	ano	NT
1	ne	T	1	ano	NT
jiná	ne	T	≠ 1	ne	T

Korelace:

- If b1 NT then b2 NT
- If b2 T then b1 T

Pozn: (NT: Not Taken, T: Taken)
=skok se neprovede/provede
=kód se provede/neprovede

Příklad

```
if (d == 0)
    d = 1;
if (d == 1)
    . . .
}
```

Předpokládejme, že R1 obsahuje hodnotu d

```
        BNEZ    R1, L1      // b1
        DADDIU  R1, R0, #1
L1:     DSUBUI  R3, R1, #1
        BNEZ    R3, L2      // b2
        . . .
L2:     . . .
```

- Budeme zkoumat, jak by n-bitový prediktor zpracoval tuto posloupnost
- Existuje celá řada možných provedení závisejících na tom, jakou hodnotu nabude d a kdy
- Učiníme několik zjednodušujících předpokladů
  - Nechť se posloupnost opakuje vícenásobně
  - Nechť hodnota d alternuje mezi 0 a 2
  - Všechna ostatní větvení budeme ignorovat

Příklad

Výchozí hodnota d	d == 0?	b1	Hodnota d před b2	d == 1?	b2
0	ano	NT	1	ano	NT
1	ne	T	1	ano	NT
jiná	ne	T	≠ 1	ne	T

- S posloupností hodnot d: 2, 0, 2, 0, 2, 0, ...
- Dostaneme následující chování skoku

b1	b2	b1	b2	b1	b2	b1	b2	b1	...
T	T	NT	NT	T	T	NT	NT	T	...

Příklad : 1-bitový prediktor

b1	b2	b1	b2	b1	b2	b1	b2	b1	...
T	T	NT	NT	T	T	NT	NT	T	...

- Předpokládejme 1-bitový prediktor (pro každý skok)
- Každé větvení má svůj prediktor
  - protože větvení nejsou dostatečně vzdálena, mají n dolních bitů stejných
- Proto každý skok sleduje posloupnost: T, NT, T, NT, T, NT, ...
- Předpokládáme-li 1-bitový prediktor inicializovaný na NT, potom KAŽDÉ VĚTVENÍ JE PREDIKOVÁNO ŠPATNĚ

Příklad: 1-bitový prediktor

```
if (d==0) d=1;
if (d==1) ….
```

```
     BNEZ   R1, L1    ;větvení b1 (d != 0)
     ADD    R1, R0, #1    ;(d = 1)
L1:  SUBI   R3, R1, #1
     BNEZ   R3, L2    ;větvení b2 (d != 1)
     …..
L2:
```

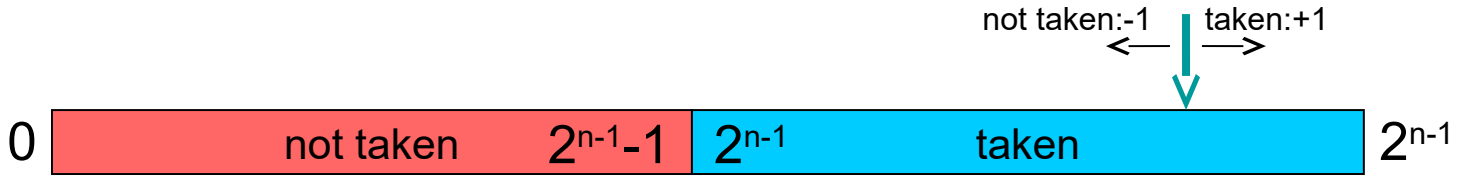
Predikce větvení b1

Aktualizace větvení b1

d=?	b1 Pre	b1 Act	New b1	b2 Pre	b2 Act	New b2
2	NT	T	T	NT	T	T
0	T	NT	NT	T	NT	NT
2	NT	T	T	NT	T	T
0	T	NT	NT	T	NT	NT

Příklad : n-bitový prediktor

- Každý skok sleduje posloupnost: T, NT, T, NT, T, NT, ...
- Uvažujme n-bitový prediktor



- Ať je prediktor inicializován jakkoli < $2^{n-1}-1$ nebo > 2^{n-1}, stejně je polovina větvení predikována špatně
  - Predikce je vždy stejná
- Je-li prediktor inicializován na $2^{n-1}-1$, potom všechna větvení budou predikována chybně
  - Prediktor osciluje v protifázi s větveními!
- Je-li prediktor inicializován na 2^{n-1}, potom je špatně predikována polovina větvení

- Závěr: n-bitový prediktor nepracuje právě nejlépe!

Predikce větvení s korelací

- Důvodem, proč je predikce slabá je to, že skoky jsou korelované!
  - If b2 taken then b1 taken
  - If b1 not taken then b2 not taken
- Základní myšlenka: Každý skok bude mít dva bity
  - Jeden bit je použit, když poslední provedený skok nebyl proveden
  - Jeden bit je použit, když poslední provedený skok byl proveden
  - S oběma bity je zacházeno, jako s normálními 1-bitovými prediktory
- Poznámka - “poslední provedený skok” typicky NENÍ skok, který zkoušíme predikovat
- Notace:
  - Aktuální stav prediktoru X / Y
    - X: použit, když poslední skok nebyl proveden
    - Y: použit, když poslední skok byl proveden
  - Příklad:
    - T / NT: nebyl-li poslední skok proveden, pak je skok predikován kladně; nebyl-li poslední skok proveden, pak je skok predikován negativně

Příklad : Prediktor s korelací

b1	b2	b1	b2	b1	b2	b1	b2	b1	...
T	T	NT	NT	T	T	NT	NT	T	...

- Uvažujme větvení b1 a předpokládejme, že prediktor s korelací je inicializován na NT / NT

b1 predictor	b1 prediction	b1 action	b1 predictor	b2 action
NT / NT	NT	T	T / NT	T
T / NT	NT	NT	T / NT	NT
T / NT	T	T	T / NT	T
T / NT	NT	NT	...	...

- Téměř všechna větvení jsou predikována správně!

Příklad : Prediktor s korelací

b1	b2	b1	b2	b1	b2	b1	b2	b1	...
T	T	NT	NT	T	T	NT	NT	T	...

- Co kdybychom provedli jinou aktualizaci prediktoru ?

b1 predictor	b1 prediction	b1 action	b1 predictor	b2 action
NT / NT	NT	T	NT / T	T
NT / T	T	NT	NT / NT	NT
NT / NT	NT	T	T / NT	T
T / NT	NT	NT	...	...

- Podobné chování po prvých několika iteracích: T / NT

Predikce s korelací

- Prediktor s korelací jsme pozorovali jen v tomto specifickém případě
- Je velmi jednoduchý a je speciálním případem obecnějšího (m,n) prediktoru
  - Využívá chování posledních m větvení
  - Vybírá z 2^m různých prediktorů
  - Každý z těchto prediktorů je n-bitový prediktor
- Vyžaduje mnohem více hardware, ale princip je úplně stejný

„Potrhlosti“ predikce větvení

- V reálných systémech jsou často implementovány násobné prediktory
- Nadřazený “metaprediktor” pak používá algoritmus, pomocí kterého vybírá, který prediktor se právě použije
- Nazývá se kombinovaný prediktor (tournament predictor)

Dynamická predikce skoků

- Výkon = f(úspěšnost, cena špatné predikce)

Tabulka historie větvení (BHT – branch history table): Dolní bity PC adresují indexovou tabulku 1-bitových hodnot

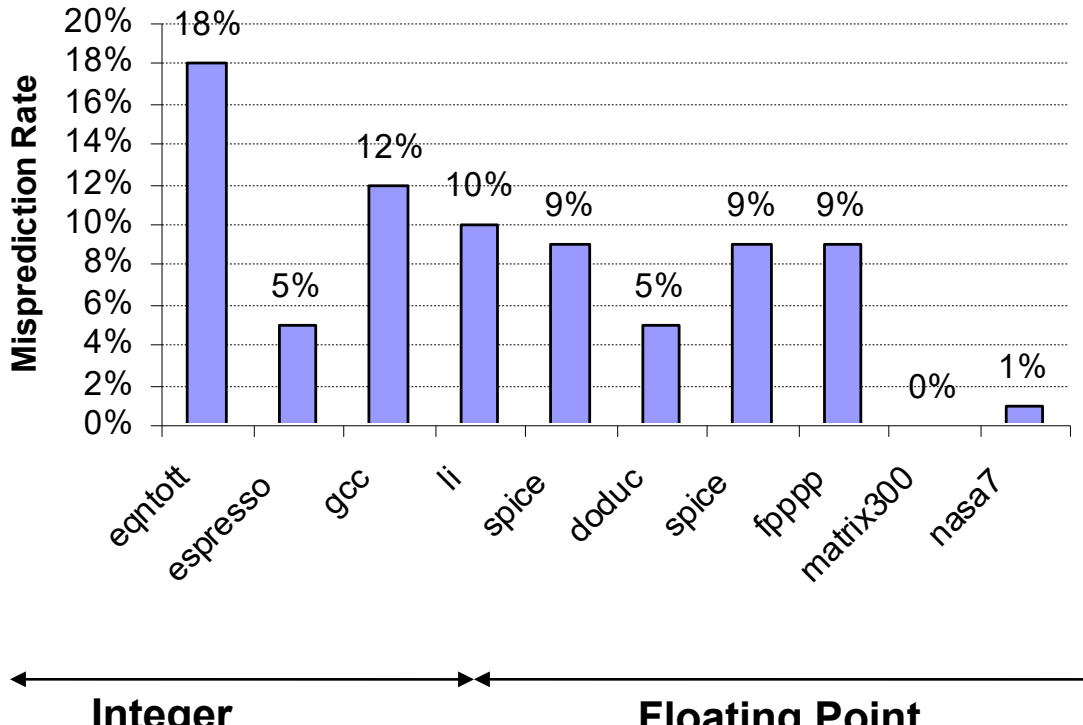
  - Říká, zda byl skok při posledním průchodu konán
  - Bez kontroly adresy

- Problém: Ve smyčce způsobí 1-bitová BHT dvě špatné predikce (střední h. je 9 iterací před opuštěním smyčky):
  - Při ukončování smyčky, kdy se odchází namísto návratu počátek smyčky jako před tím
  - Při prvním průchodu smyčkou, kdy je naplánován odchod místo standardních iterací smyčky

Účinnost použití BHT

(BHT= branch history table)

- Špatná predikce nastává kvůli:
  - Špatnému odhadu pro daný podmíněný skok
  - Při dostupu do tabulky pomocí indexu je získána historie jiného skoku
- Tabulka s 4096 položkami:



Predikce větvení s korelací

- 2-bitová predikce používá malé množství lokální informace k predikci chování skoku
- Chování bývá korelováno, lepšího výsledku můžeme dosáhnout sledováním směru vázaných skokových instrukcí. Například:

```
if (d==0)
    d = 1;
if (d==1) {
```

- Nekoná-li se prvý skok, nebude se konat ani druhý. Prediktory, které využívají k predikci chování ostatních skoků se nazývají prediktory s korelací *(correlating predictors)*, nebo-li dvouúrovňové prediktory.

Predikce větvení s korelací

- **Idea:** Zaznamenejme *m* naposledy provedených větvení jako provedené (T) nebo neprovedené (NT) a použijme tento vzorek k výběru správné n-bitové tabulky historie větvení

- Obecně, (*m*,*n*) prediktor znamená zaznamenání posledních *m* větvení k výběru mezi 2^m tabulkami historie, každá s n-bitovými čítači
  - Dříve uvedený 2-bitový prediktor je tedy prediktor typu (0,2)

- Globální historie větvení: *m*-bitový posuvný registr, obsahující T/NT status posledních *m* větvení.
- Každá položka tabulky obsahuje *m* *n*-bitových prediktorů.

Příklad: Prediktor (1,1)

Naposledy se skok:

nekonal (NT) – test prvého bitu

konal (T) – test druhého bitu

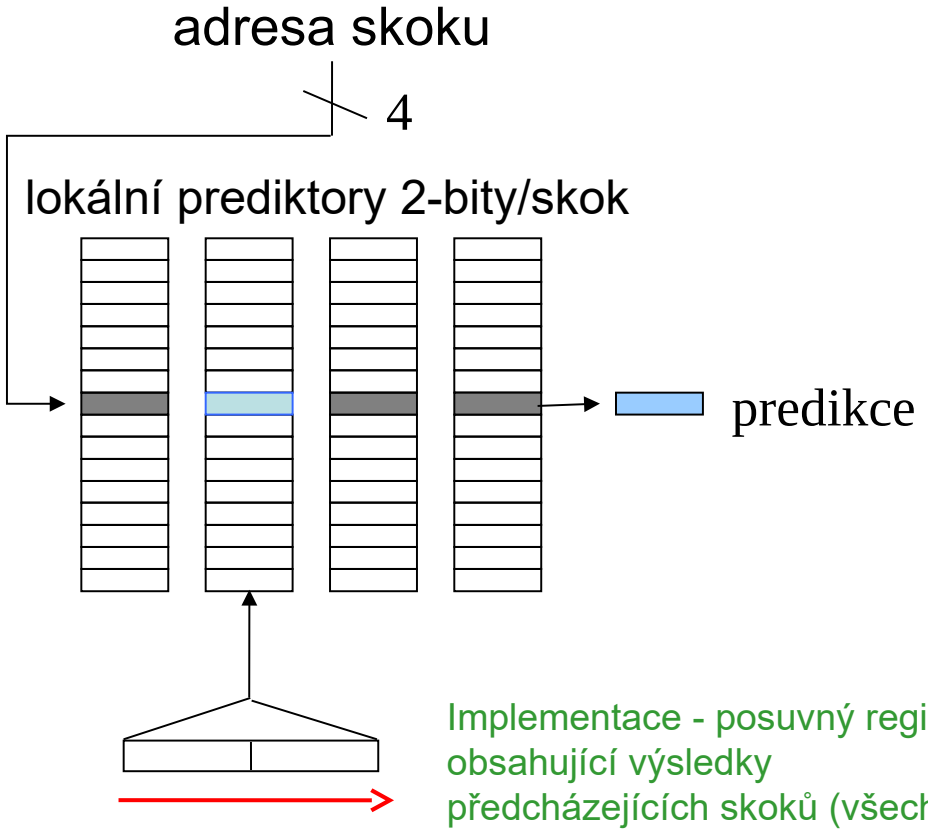
Prediction Bits	Prediction if LB NT	Prediction if LB T
NT/NT	NT	NT
NT/T	NT	T
T/NT	T	NT
T/T	T	T

d=?	b1 Pre	b1 Act	New b1	b2 Pre	b2 Act	New b2
2	**NT**/NT	T	T/NT	NT/**NT**	T	NT/T
0	T/**NT**	NT	T/NT	**NT**/T	NT	NT/T
2	**T**/NT	T	T/NT	NT/**T**	T	NT/T
0	T/**NT**	NT	T/NT	**NT**/T	NT	NT/T

Predikce větvení s korelací

Chování (T/NT) naposledy provedených skoků je vztaženo na chování příštího skoku (stejně jako historie tohoto skoku).

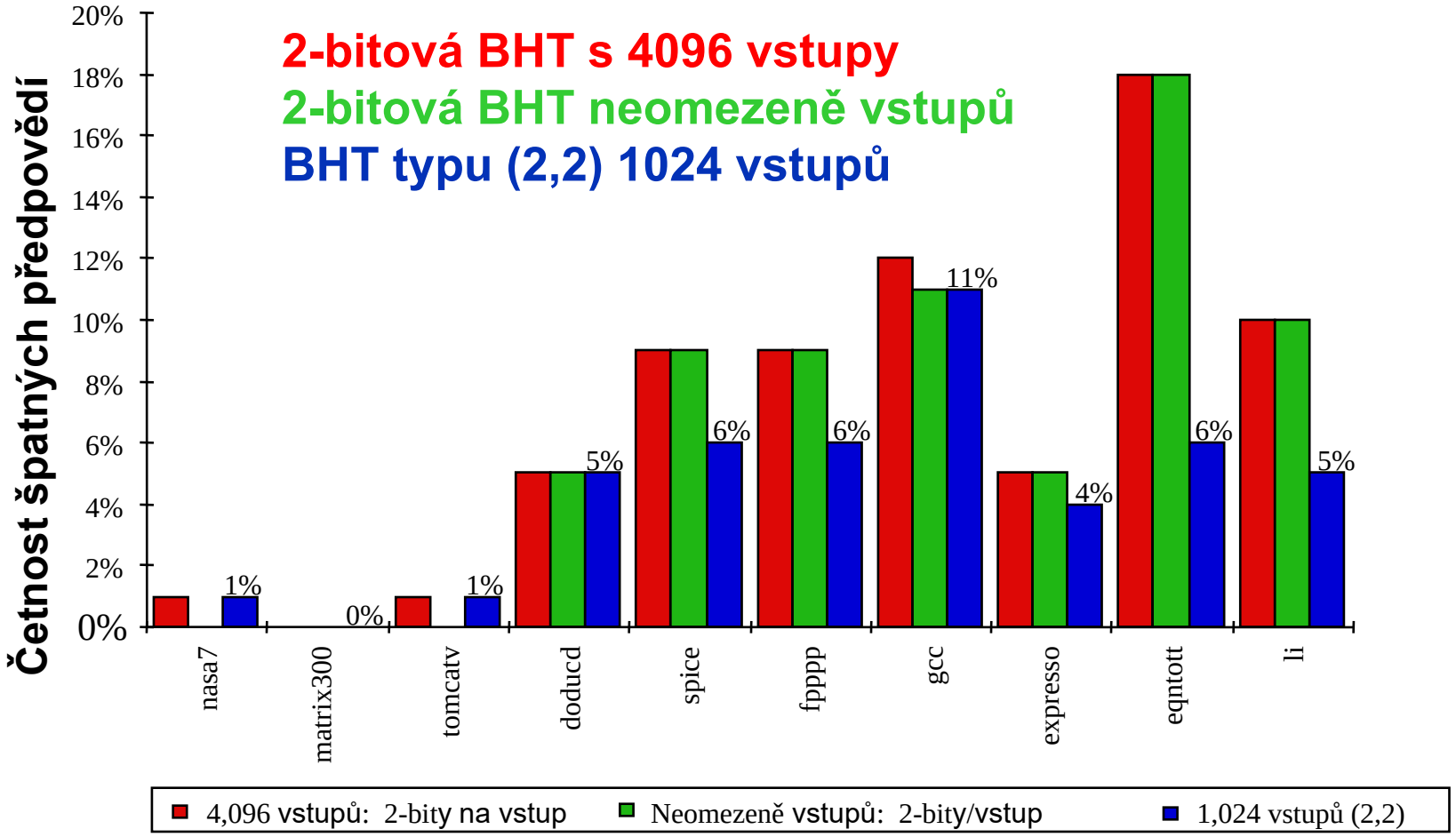
- Chování naposledy prováděného větvení vybírá mezi 4 predikcemi příštího skoku a provede aktualizaci predikce



- (2,2) prediktor: 2 bity globální, 2 bity lokální

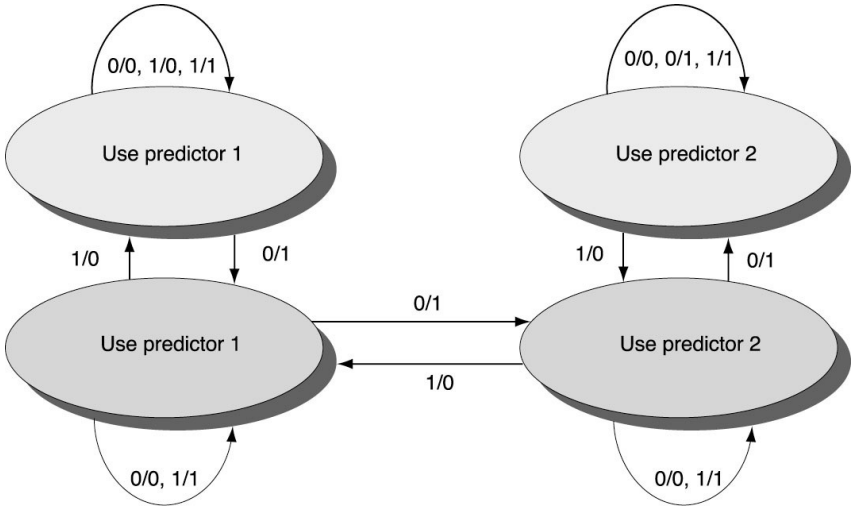
zde 2-bitová globální historie větvení
01 = T a pak NT

Přesnost různých schémat



Kombinované prediktory (tournament)

- Víceúrovňový prediktor větvení
- Obvyklá volba mezi globálními a lokálními prediktory



- Použití *n*-bitových čítačů se saturací pro výběr mezi prediktory (čítač se saturací: **Čítač, který nepřetéká a zastaví se na maximální hodnotě.)**

Dvoubitový saturační čítač jako selektor prediktoru

- Oba prediktory jsou vyhodnocovány podle správnosti výsledku, bez ohledu na to, který byl aktuálně použit k vyhodnocení skoku.
- Dávají-li oba prediktory 1 & 2 správný výsledek, selektor prediktoru není aktualizován.
- Nedávají-li oba prediktory 1 & 2 správný výsledek, selektor prediktoru také není aktualizován.
- Dává-li prediktor 1 správný výsledek a prediktor 2 špatný, 2-bitový selektor prediktoru je inkrementován (se saturací).
- Dává-li prediktor 1 nesprávný výsledek a prediktor 2 správný, je 2-bitový selektor prediktoru dekrementován (se saturací).
- Výběr prediktoru pro aktuální instrukci větvení se provádí podle hodnoty MSb selektoru prediktorů. Je-li MSb roven 1, použije se predikce 1. Je-li MSb roven 0, použije se predikce 2 .

Kombinované prediktory

„Tournament Predictors“

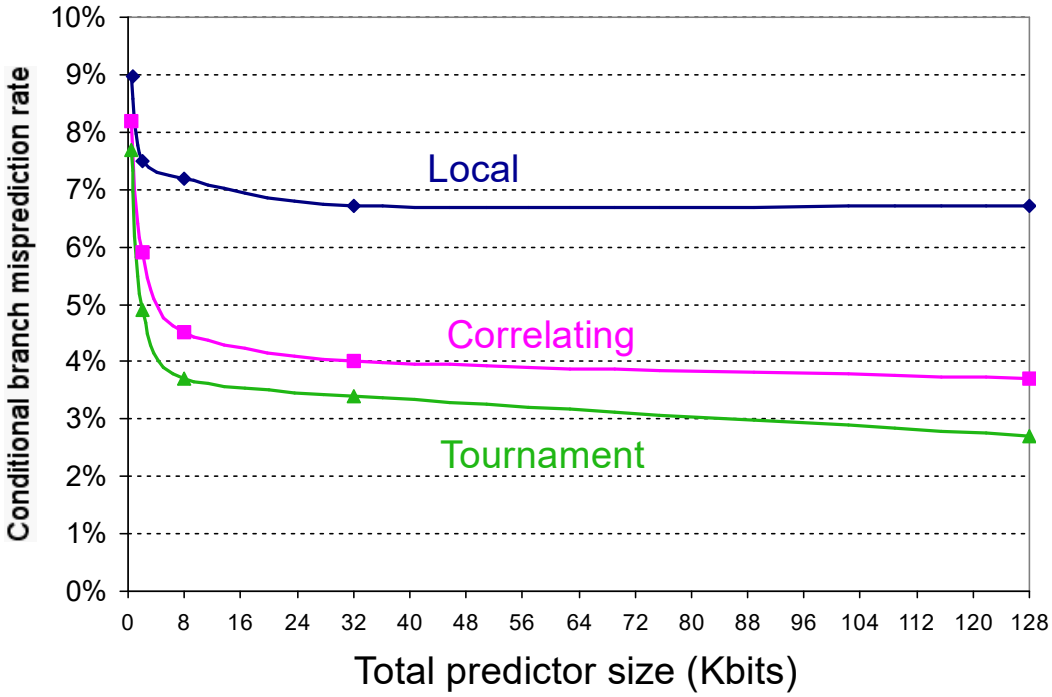
Kombinovaný prediktor používá např. 4K 2-bitových saturačních čítačů, indexovaných lokální adresou větvení. Výběr z variant:

- Globální prediktor
  - 4K položek, indexováno historií posledních 12-ti větvení ($2^{12} = 4K$)
  - Každá položka je standardní 2-bitový prediktor
- Lokální prediktor
  - Lokální tabulka historie: 1024 10-bitových položek se záznamem posledních 10-ti větvení, indexováno adresou větvení
  - Vzorek posledních 10 výskytů daného konkrétního skoku (pole 10 bitů) je použit jako index do tabulky s 1K 2-bitových prediktorů

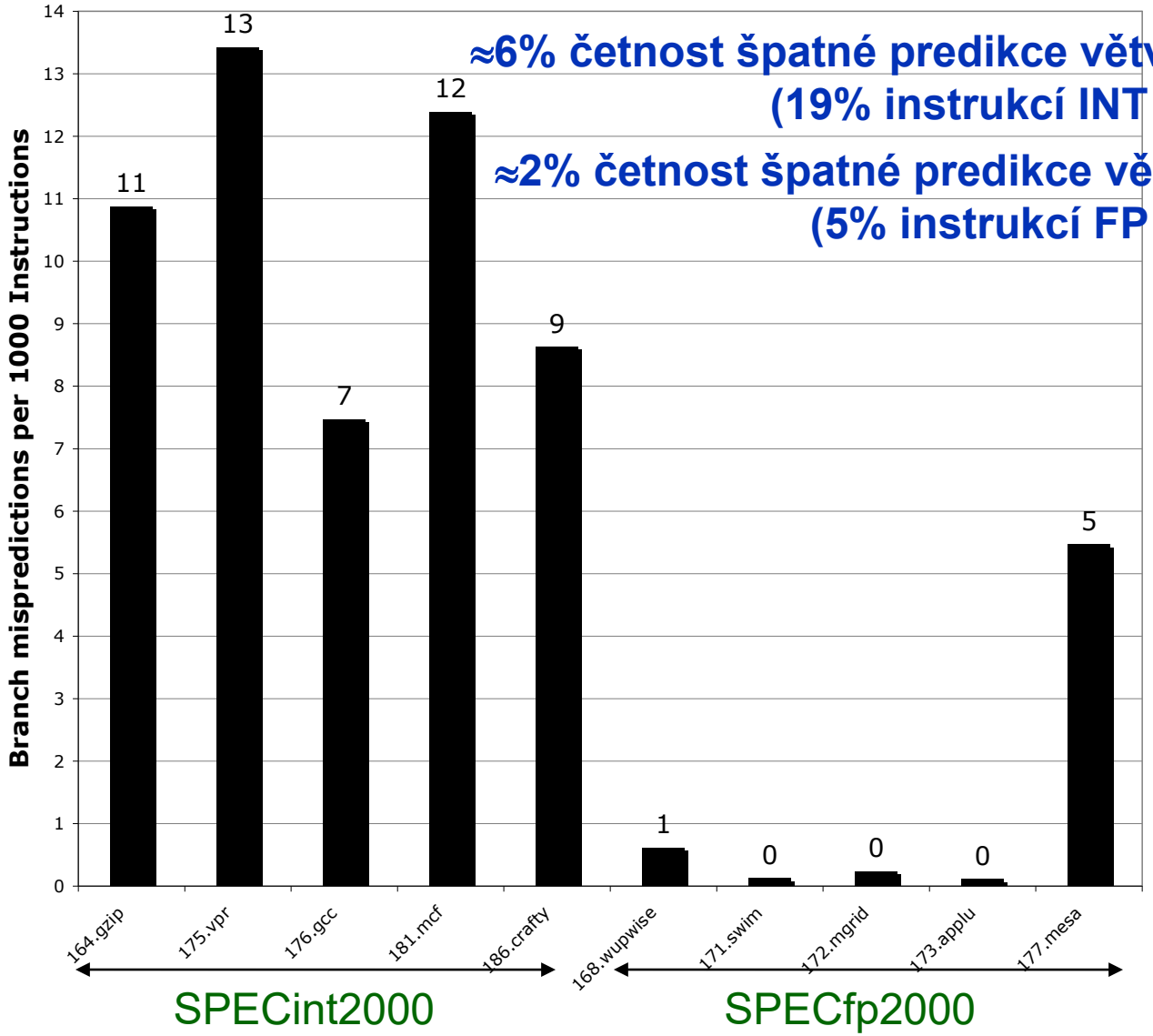
Porovnání prediktorů

- Výhodou kombinovaného prediktoru je schopnost vybrat správnou strategii predikce pro daný podmíněný skok.
  - Zvláště důležité pro integer benchmarky.
  - Typický kombinovaný prediktor vybírá globální strategii predikce ve 40% doby pro SPEC integer benchmarky a méně než 15% doby pro SPEC FP benchmarky

SPEC89



Četnost chybné predikce Pentia 4



≈6% četnost špatné predikce větvení SPECint
(19% instrukcí INT jsou větvení)

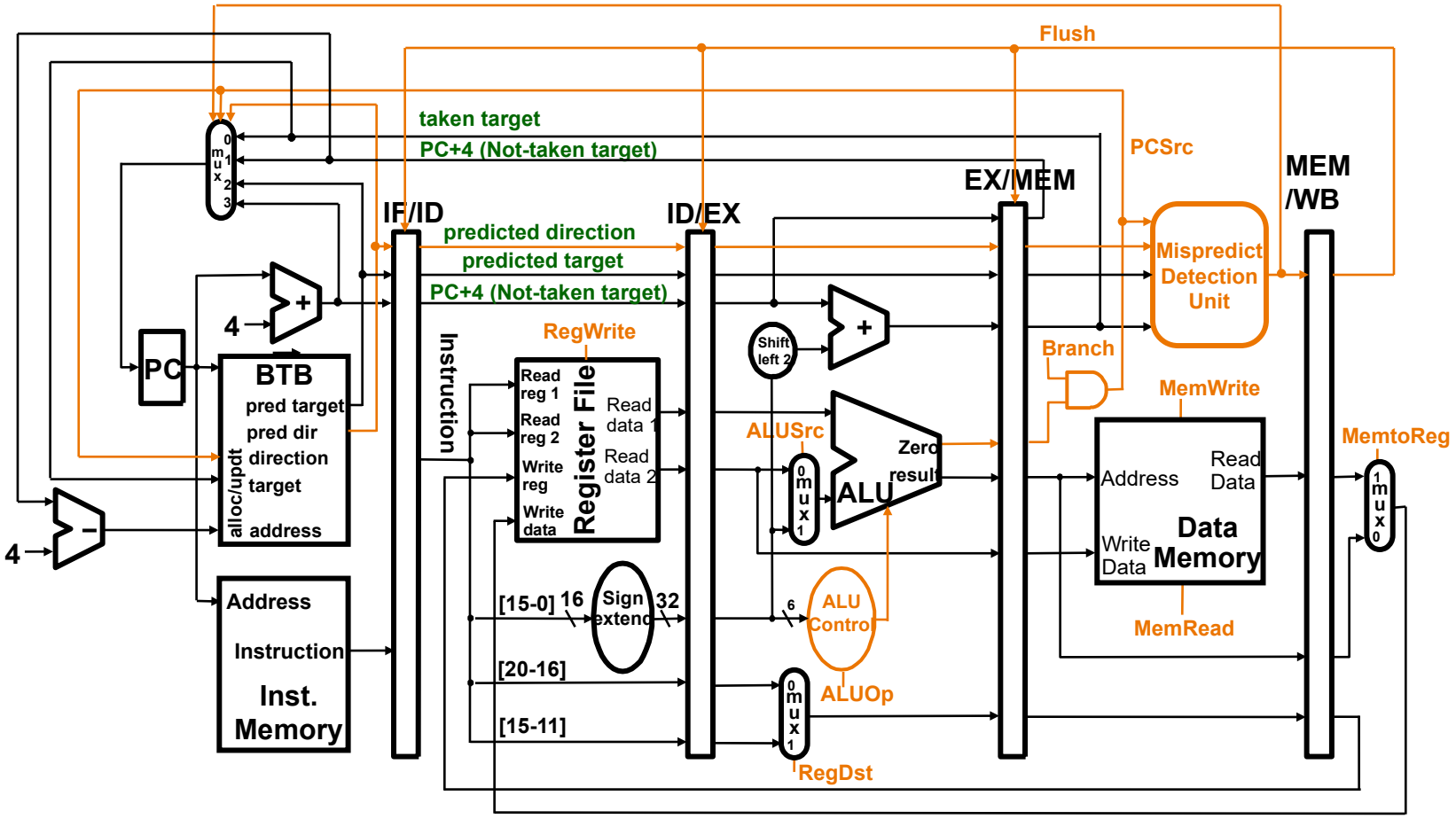
≈2% četnost špatné predikce větvení SPECfp
(5% instrukcí FP jsou větvení)

Speciální případ – návratové adresy

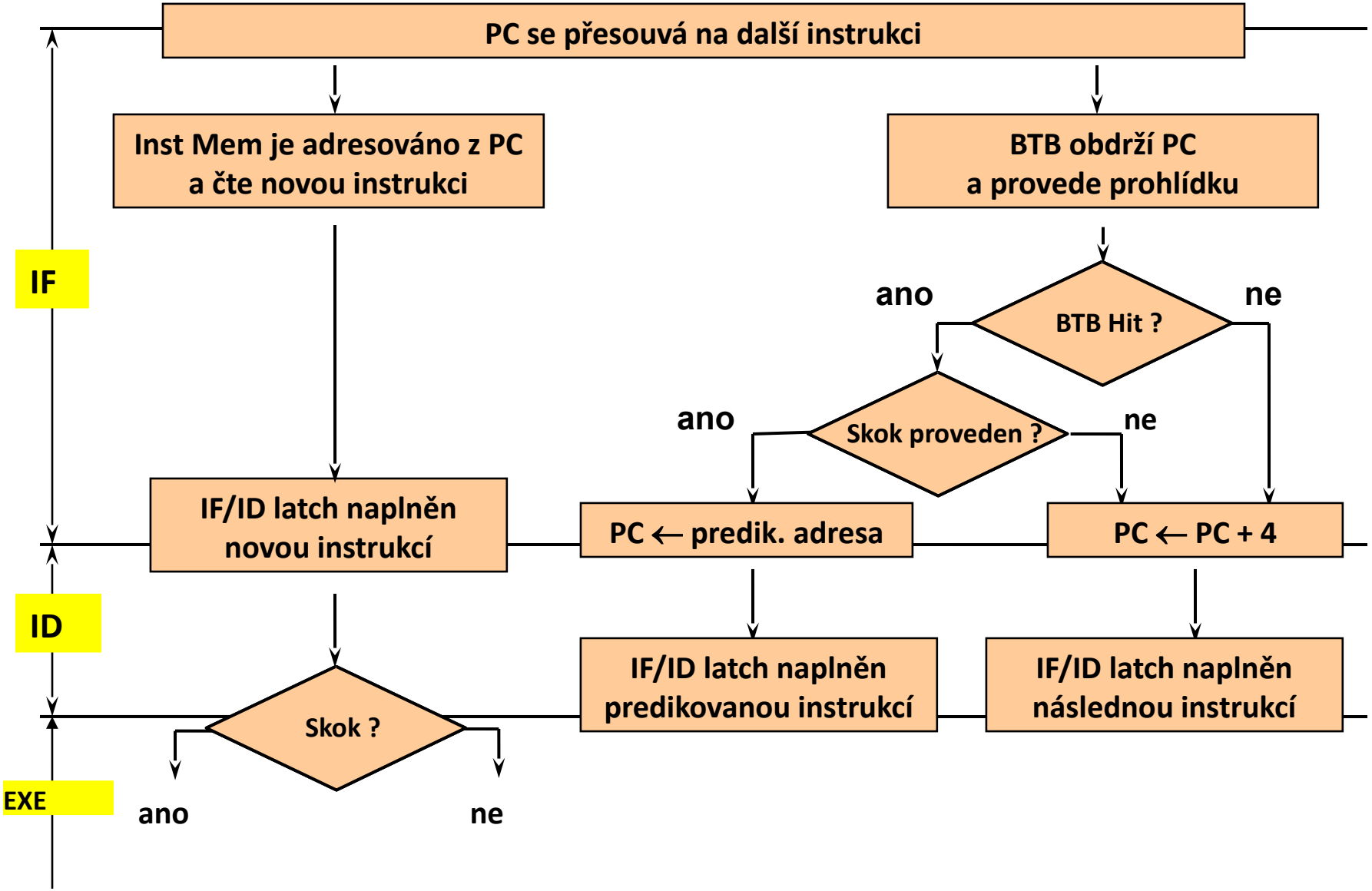
- Nepřímý skok přes obsah registru se obtížně predikuje.
- SPEC89 používá 85% takových skoků pro návrat z procedury
- Protože se procedury řeší zásobníkem, i tady se návratové adresy ukládají do malého bufferu, který pracuje jako zásobník: 8 až 16 položek postačí na podstatné snížení neúspěšného odhadu

Doplnění BTB do pipeline

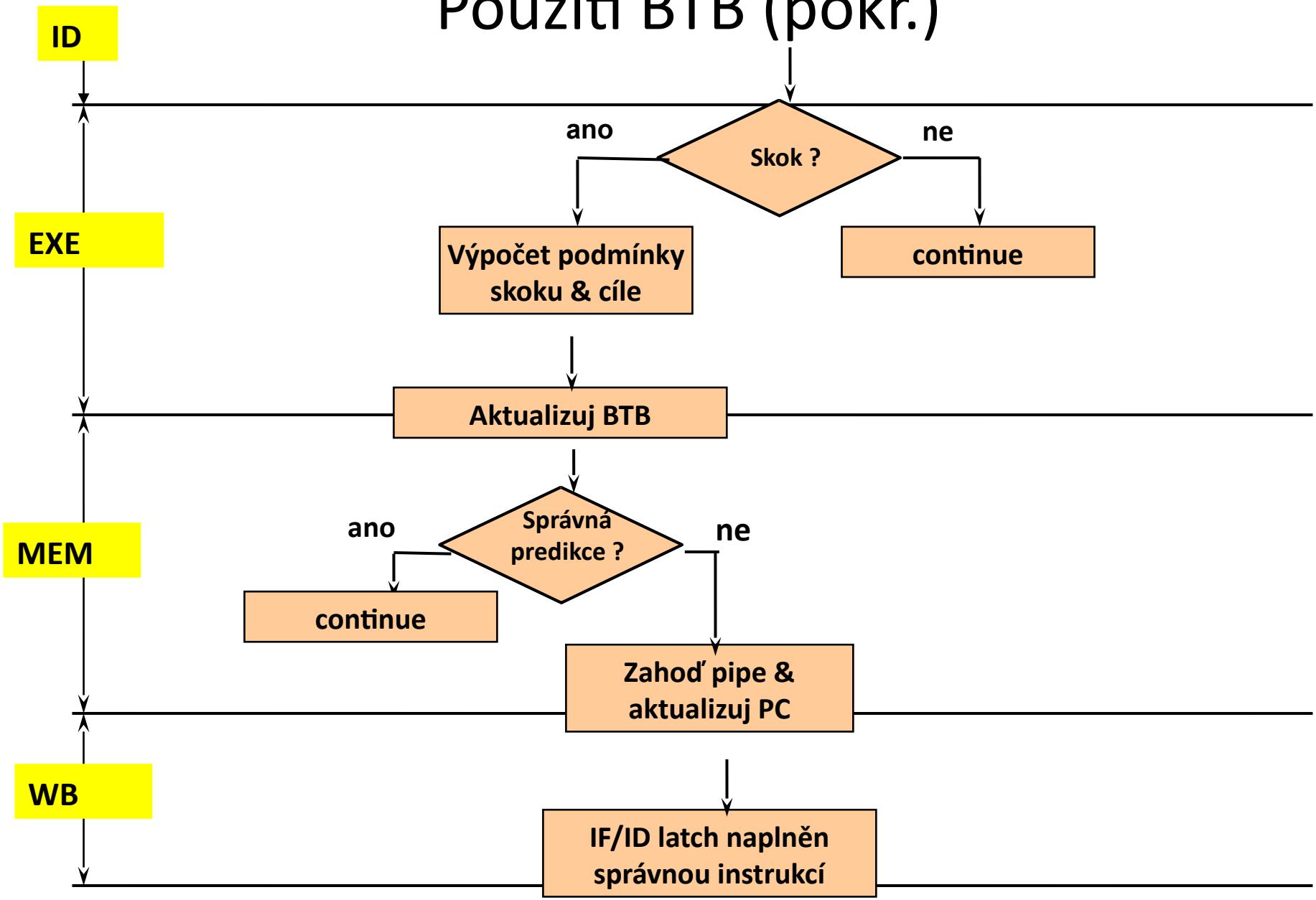
(BTB = branch targer buffer)



Použití BTB


PC se přesouvá na další instrukci
Inst Mem je adresováno z PC
a čte novou instrukci
BTB obdrží PC
a provede prohlídku
IF
ano
BTB Hit ?
ne
ano
Skok proveden ?
ne
IF/ID latch naplněn
novou instrukcí
PC ← predik. adresa
PC ← PC + 4
ID
IF/ID latch naplněn
predikovanou instrukcí
IF/ID latch naplněn
následnou instrukcí
Skok ?
EXE
ano
ne

Použití BTB (pokr.)

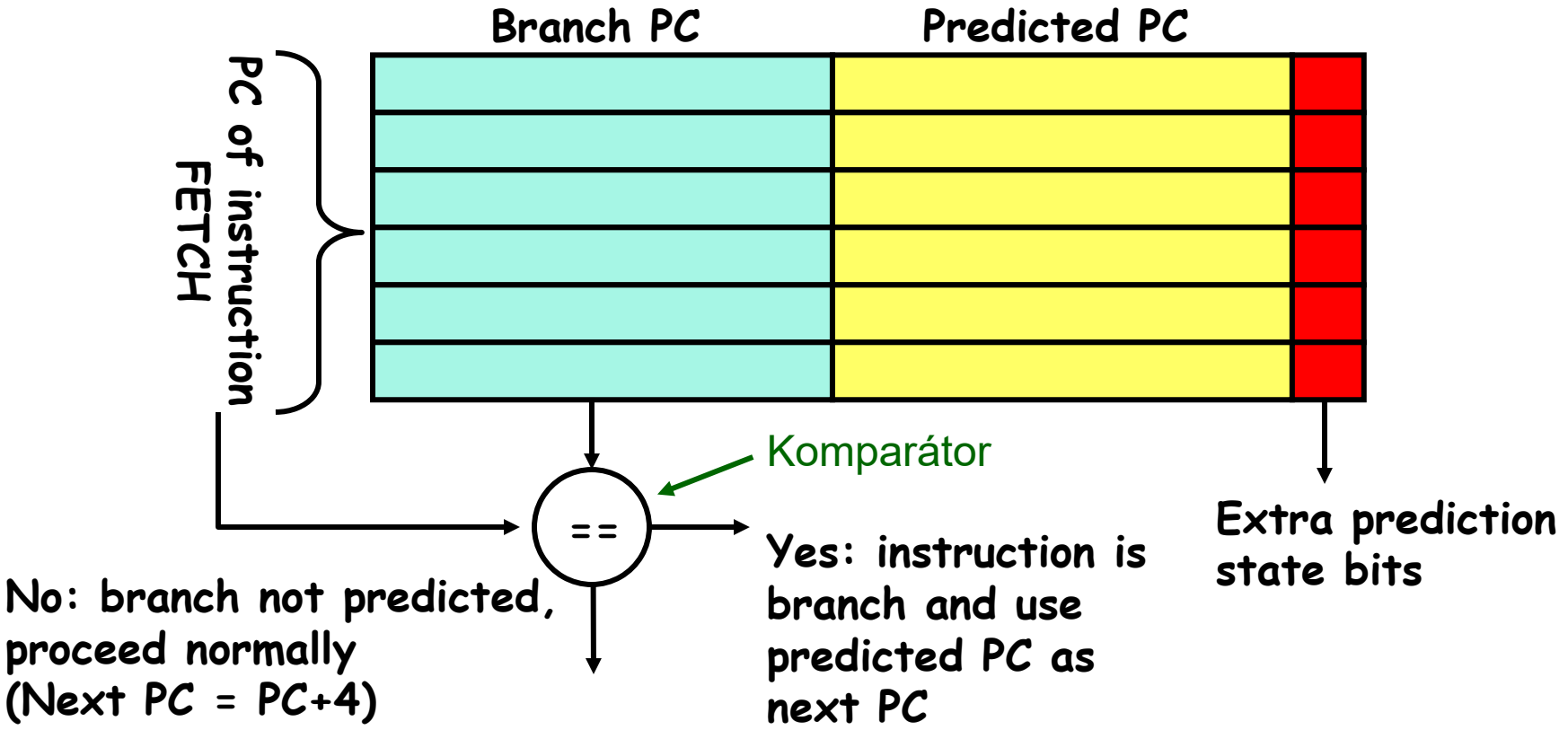


Cache pro cíle skoků (BTB)

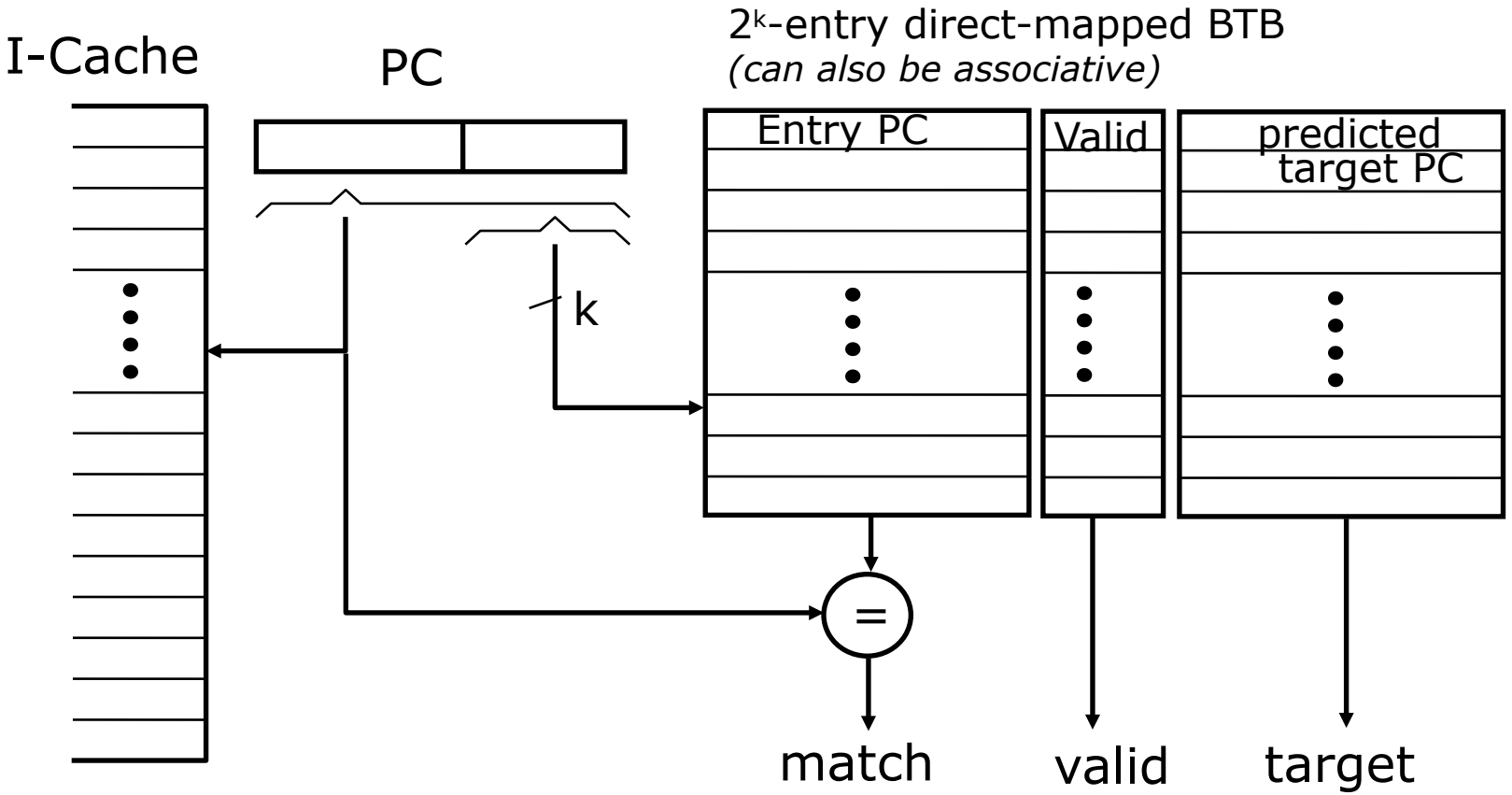
- Výpočet adresy cíle skoku je „drahý“ a zpožďuje načítání instrukcí.
- BTB ukládá PC stejným způsobem jako cache
- PC podmíněného skoku je odeslán do BTB
- Je-li nalezena shoda, vrací se odpovídající predikovaná hodnota PC (adresa cíle skoku)
- Koná-li se predikované větvení, načítání instrukcí pokračuje od adresy, kam ukazuje dodaná hodnota PC

Adresa ve stejnou dobu jako predikce

- Branch Target Buffer (BTB): Adresa - index skoku pro získání predikce AND adresy cíle skoku (když se koná)
  - Poznámka: shodu musí ověřit nyní, protože nesmí použít špatnou cílovou adresu



Buffer pro cíle skoků (BTB)



- V BTB je uložena adresa skoku a cíl skoku
- Z adresy PC+4 se načítá instrukce, nenajde-li se shoda
- V BTB se ukládají jen *provedená větvení a skoky*
- Příští PC je určeno *dřív* než je skok načten a dekódován

Dynamická predikce skoků - souhrn

- Predikce se stala důležitou součástí provádění výpočtu
- Tabulka historie větvení (BHT) : 2 bity pro dobrou predikci smyček
- Korelace: Naposledy provedené větvení je korelováno s následujícím
  - Buď jiná instrukce větvení (GA)
  - Nebo další provedení téhož větvení (PA)
- Kombinované prediktory zohledňují další úroveň využitím více prediktorů
  - jeden obvykle využívá globální informace, druhý je založen na analýze lokálního chování. Navzájem jsou kombinovány selektorem.
  - V roce 2006 byly používány kombinované prediktory s kapacitou paměti ≈ 30K bitů, např. v procesorech jako PowerPC a Pentium 4
- Buffer pro cíle skoků (Branch Target Buffer): zahrnuje adresu větvení & predikci