# Zwiększanie wydajności pamięci podręcznych

Przemek Sierociński

26 listopada 2014

# Czas stosowania

Metody zwiększania wydajności pamięci podręcznych ze względu na moment, w którym się je stosuje:

- on-line:
  - w trakcie wykonania programu, np. profilując zachowanie programu
- off-line:
  - przed uruchomieniem programu (w trakcie jego pisania, kompilowania)
- mieszane:
  - na przykład wykorzystanie instrukcji warunkowego załadowania jak w HPD-PD (Intel Itanium)

# Charakter optymalizacji

Pomysły optymalizacji czasu dostępu do danych można podzielić na kategorie:

- Strategie wymiany bloków
- Strategie prefetchingu
- Optymalizacje lokalności

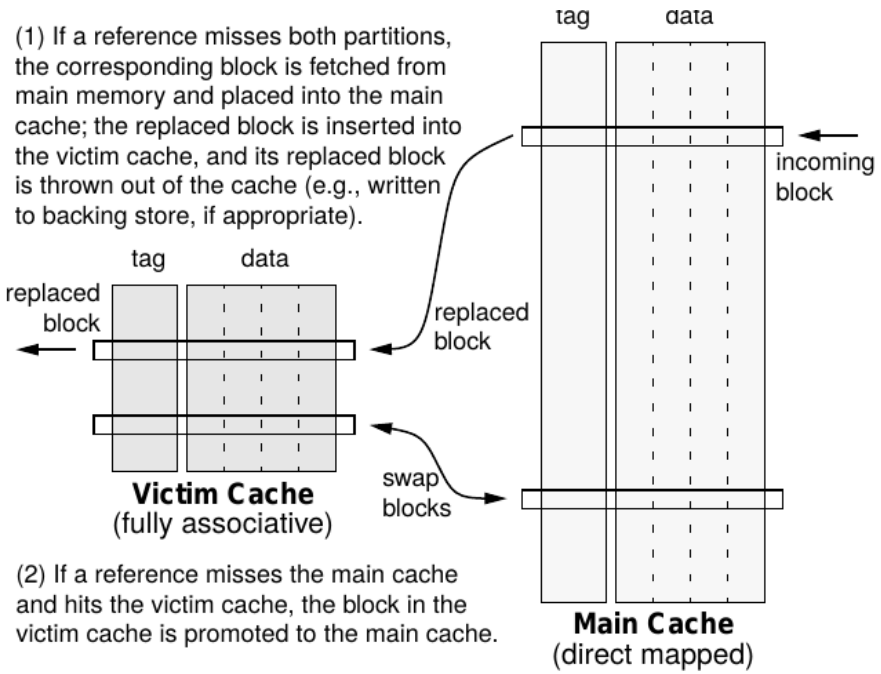
# Strategie wymiany bloków

- LRU
- Pseudo-LRU
- LFU
- MRU
- Random
- ...

# Victim Cache

- odpytywany równocześnie z L1
- zawartość całkowicie rozłączna z L1
- fully associative
- jeśli blok ma być usunięty z L1, trafia do victim cache (zastąpiony przez niego blok z victim zostaje wyrzucony)
- jeśli blok nie jest znaleziony w L1, ale jest w victim, to w nagrode jest wciągany spowrotem do L1 (zwykle kosztem nowej ofiary z L1)
- świetnie kompensuje dostęp do danych prezentujących lokalność czasową, ale skonfliktowanych w L1

# Victim Cache

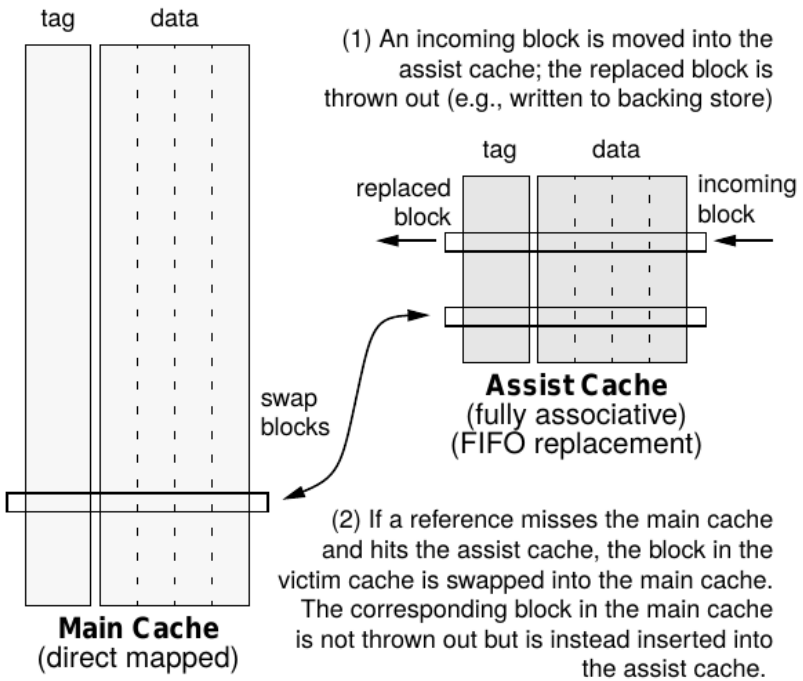



Rysunek: Schemat victim cache.

# Assist Cache

- odpytywany równocześnie z L1
- zawartość całkowicie rozłączna z L1
- fully associative
- bloki zamiast do L1 najpierw trafiają do assist cache (stare bloki z assist są usuwane)
- kiedy blok nie jest znaleziony w L1, ale jest w assist, to w nagrode zostaje przeniesiony do L1 (zwykle zamieniony z jakimś)
- dobrze zaspokaja potrzeby wiekszości danych prezentujących lokalność przestrzenna, zapewniając lepszą obsługę danych o lokalności czasowej

# Assist Cache

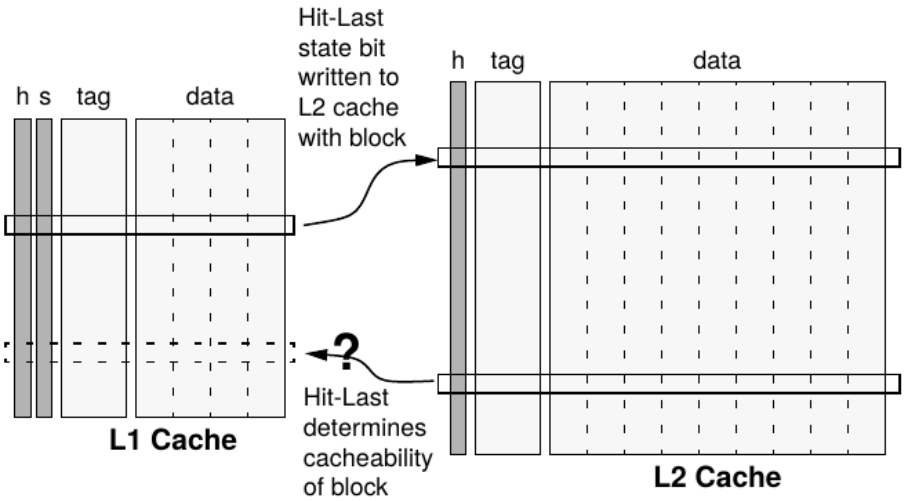


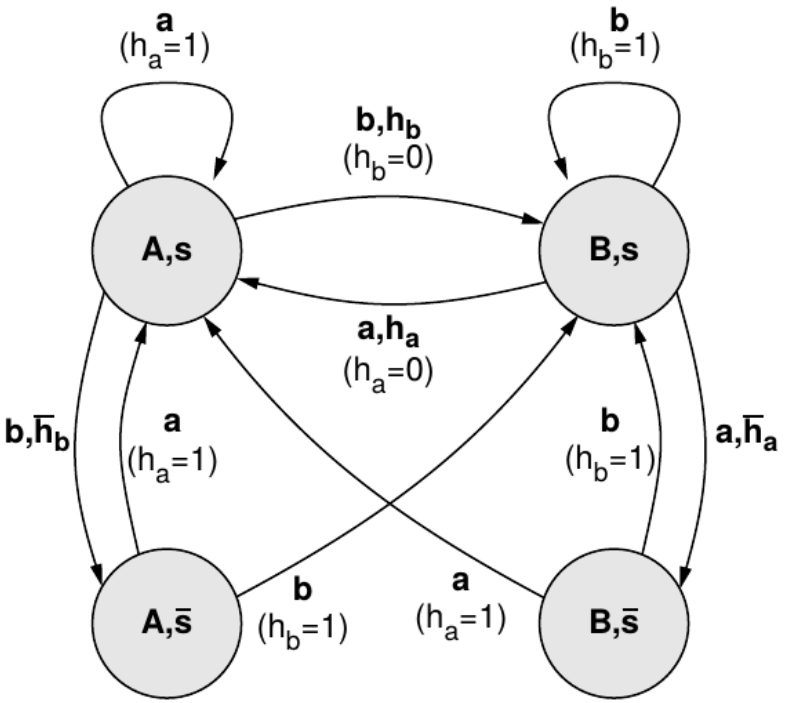

Rysunek: Schemat assist cache.

# Dynamiczne wykluczanie (McFarlings)

- motywacja: sprzętowe rozwiązanie dla instrukcji kolidujących w cache
- dodatkowe bity w cache
  - sticky - oznacza, że blok jest cenny i nie należy się go odrazu pozbywać
  - hit-last - oznacza, że blok był użyteczny ostatnim razem i warto go przyjąć znowu (ta informacja wraca z danymi do pamięci podrzędnej)
- prosta maszyna stanowa (patrz następny slajd)

# Dynamiczne wykluczanie (McFarlings)



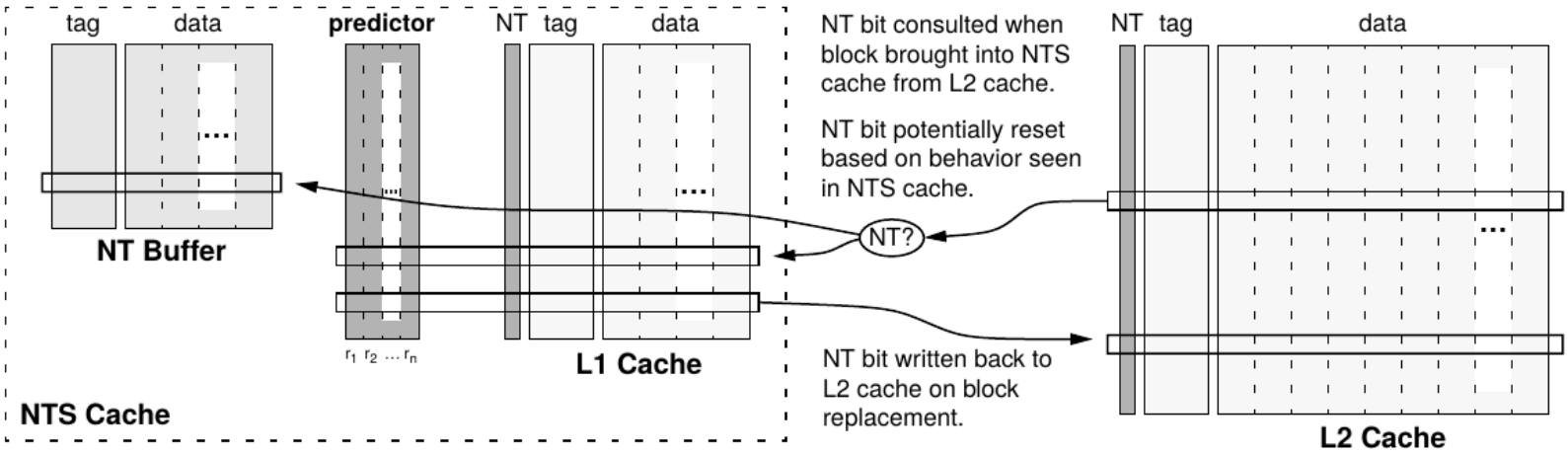




Rysunek: Schemat mechanizmu dynamic exclusion.

# Non-Temporal Streaming Cache (Rivers)

- motywacja: identyfikacja i odseparowanie danych nie przejawiających lokalności czasowej
- dodatkowy bit - Non-Temporal (NT) - zapisywany do pamięci podrzędnej
- dodatkowy cache na obok L1 (podobny do victim)
- dodatkowa kolumna licząca odwołania do poszczególnych słów w linii
- kiedy blok znajduje się w L1, jeśli nie zanotowano ponownego odwołania do żadnego z jego słów, ustawiny jest NT
- bloki, które kiedyś były w L1 i zostały odesłane do L2 z ustawionym NT po powrocie zostają skierowane do cache NT.

# Non-Temporal Streaming Cache (Rivers)

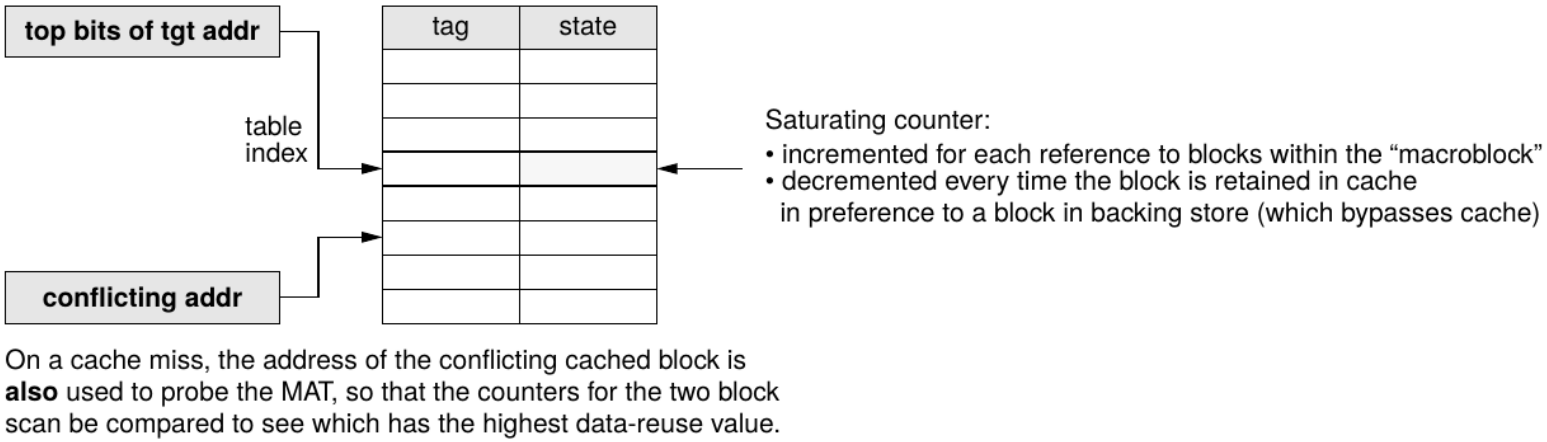



Rysunek: Schemat mechanizmu NT Streaming Cache

# Śledzenie częstości (Johnson)

- motywacja: dokładniejsze wyprofilowanie odczytów
- dodatkowy cache - memory address table (MAT)
  - każda linia w MAT opisuje makroblok, skumulowaną informacje o wielu blokach
  - zawiera licznik z nasyceniem określający częstość dostępów
  - każde odwołanie do elementu z makrobloku powiększa jego licznik
  - jeśli żądanego bloku nie ma w cache, jego licznik jest porównywany z tym, z którym ma być zamieniony i jeśli wygrał porównanie, to tak się dzieje
  - jeśli przegrał, licznik bloku pozostającego w cache jest zmniejszany

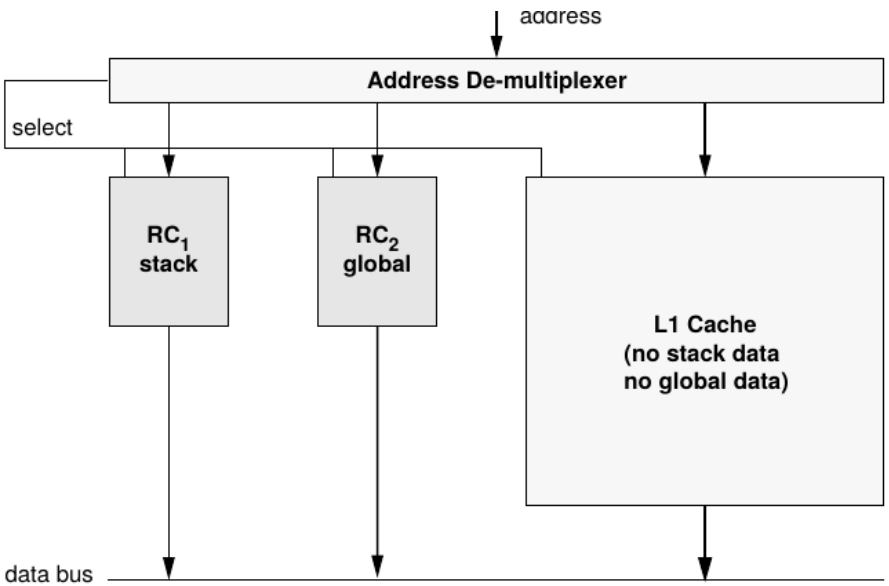
# Śledzenie częstości (Johnson)




Rysunek: MAT w mechanizmie śledzenia częstości.

# Podział ze względu na region w programie (Lee)

- motywacja: wykorzystanie faktu, że różne regiony (segmenty) programu prezentują różne schematy dostępów
- osobny cache na stos, dane globalne i całą resztę



# Prefetching on-line

Prefetching - sprowadzanie informacji do cache z wyprzedzeniem faktycznych dostępów do pamięci.
Osobne wyzwania stwarza prefetching:

- instrukcji
- danych

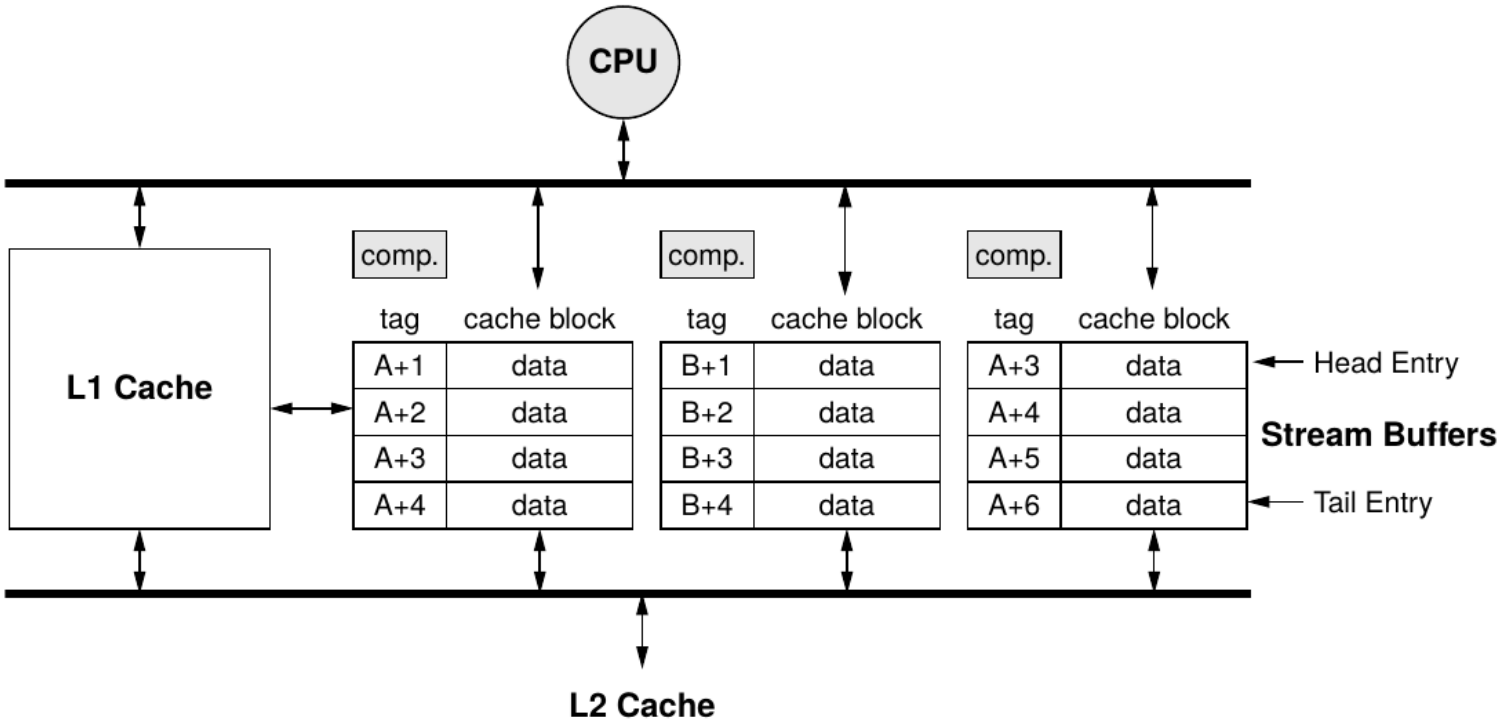
Kiedy uruchamiać prefetching?

- wykonuj prefetch za każdym razem
- wykonuj prefetch tylko przy cache-missach
- wykonuj prefetch na podstawie obserwacji wcześniejszych zachowań

# Stream buffers (Jouppi)

- kilka buforów na tym samym poziomie, co L1
- jeśli bloku nie ma w cache, a jest na szczycie kolejki buforu, to jest wciągany do cache, a bufor wysyła żądanie do pamięci o kolejny element
- jeśli elementu nie ma ani w L1, ani na szycie żadnej z kolejek, wykorzystaj jedną z kolejek do prefetchingu kolejnych elementów tego strumienia

# Stream buffers (Jouppi)

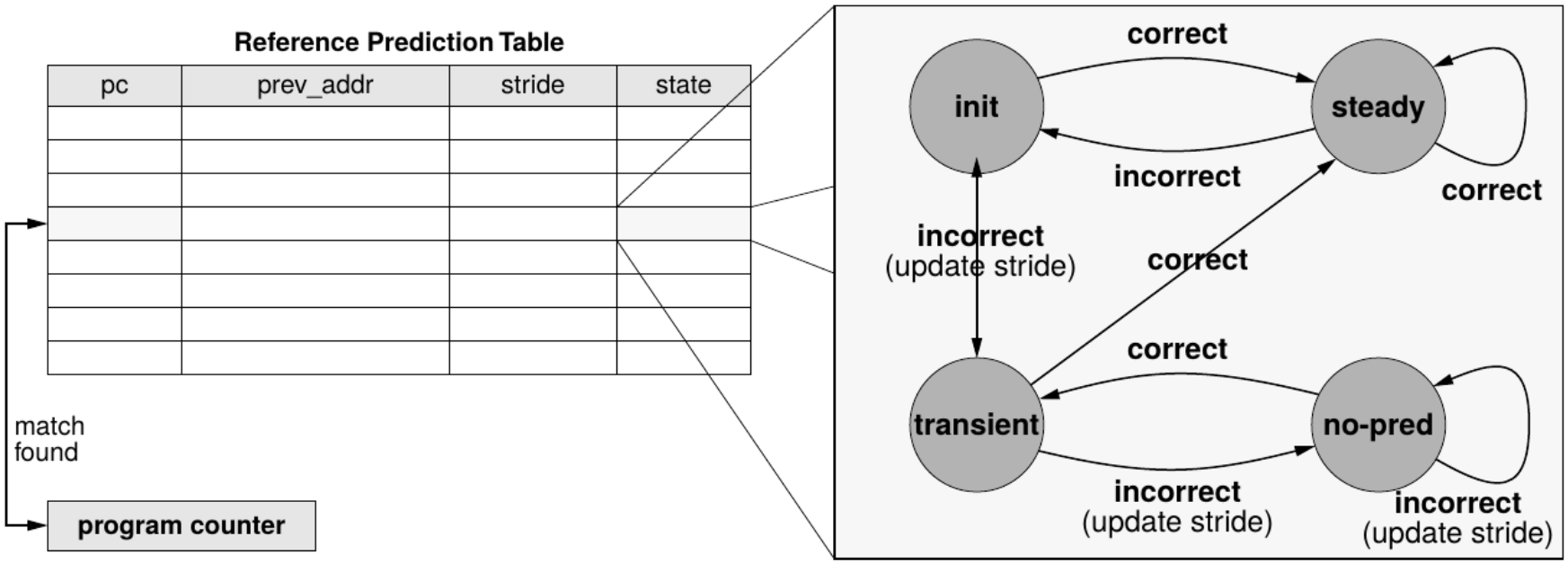



Rysunek: Schemat stream buffers.

# Reference prediction table (Baer)

- motywacja: efektywny prefetching dla sekwencji postaci: A, A+2, A+4, ...
- reference prediction table
  - indeksowana PC
  - zawiera poprzedni adres spod którego odbył się odczyt
  - odległość dwóch ostatnich odczytów
  - i czy taka odległość się już powtórzyła

# Reference prediction table (Baer)

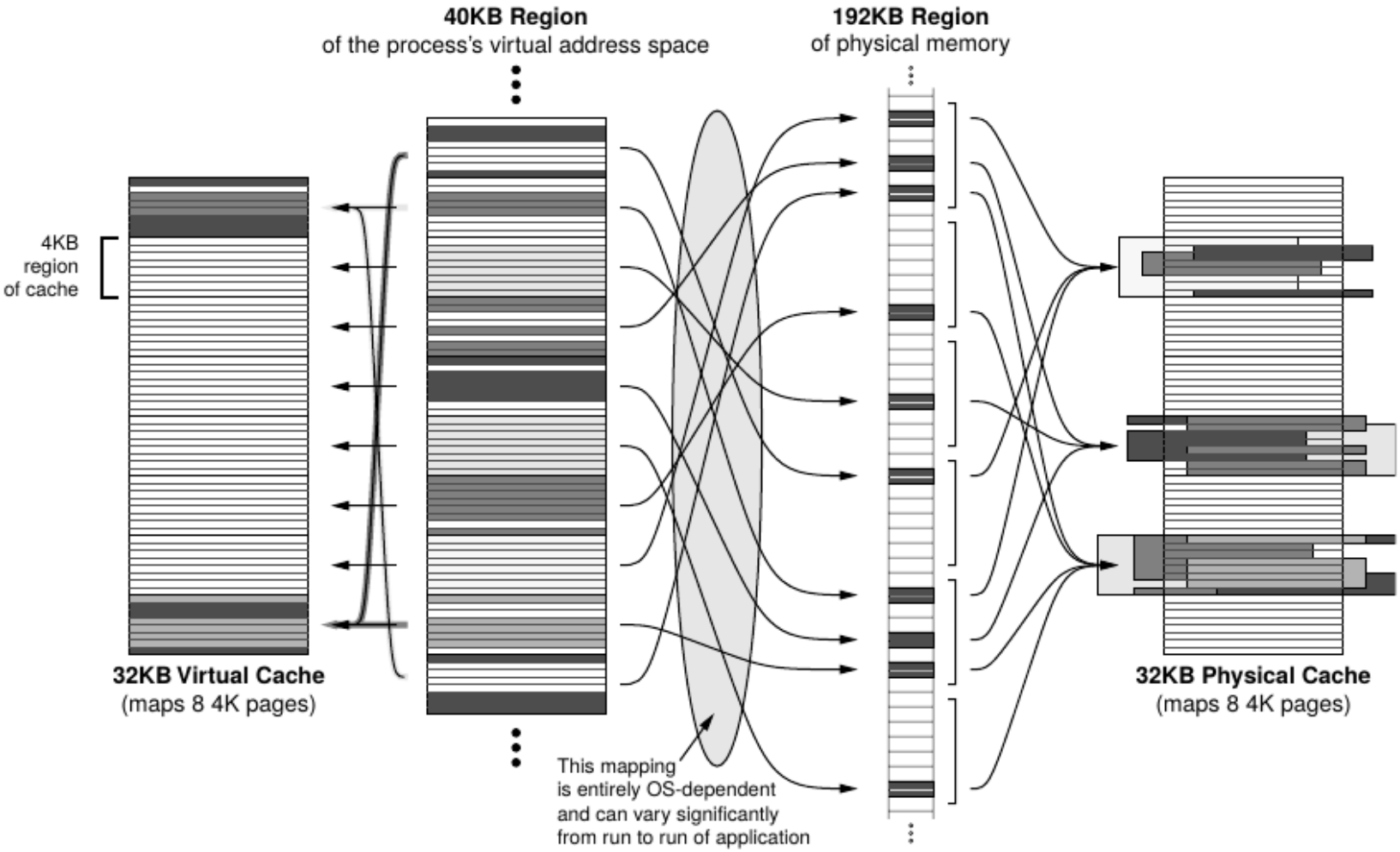



Rysunek: Schemat tablicy przewidywania odwołań.

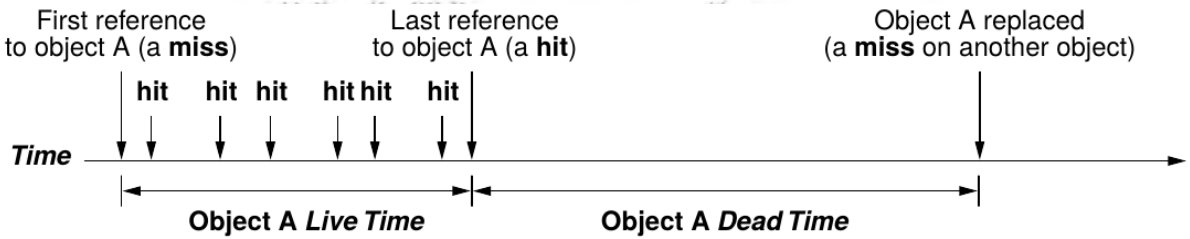
# Kolorowanie stron

- motywacja: zwiększenie hit-rate w cache (TLB slice w MIPSie)
- strategia systemu operacyjnego na przydzielanie numerów ramek do stron pamięci
- porządanym jest, żeby ostatnie bity adresu wirtualnego i fizycznego się zgadzały
- można hashować ID procesu z adresem wirtualnym, żeby zmniejszyć konflikty

# Kolorowanie stron



# Oszczędzanie energii w cache (Kaxiras)



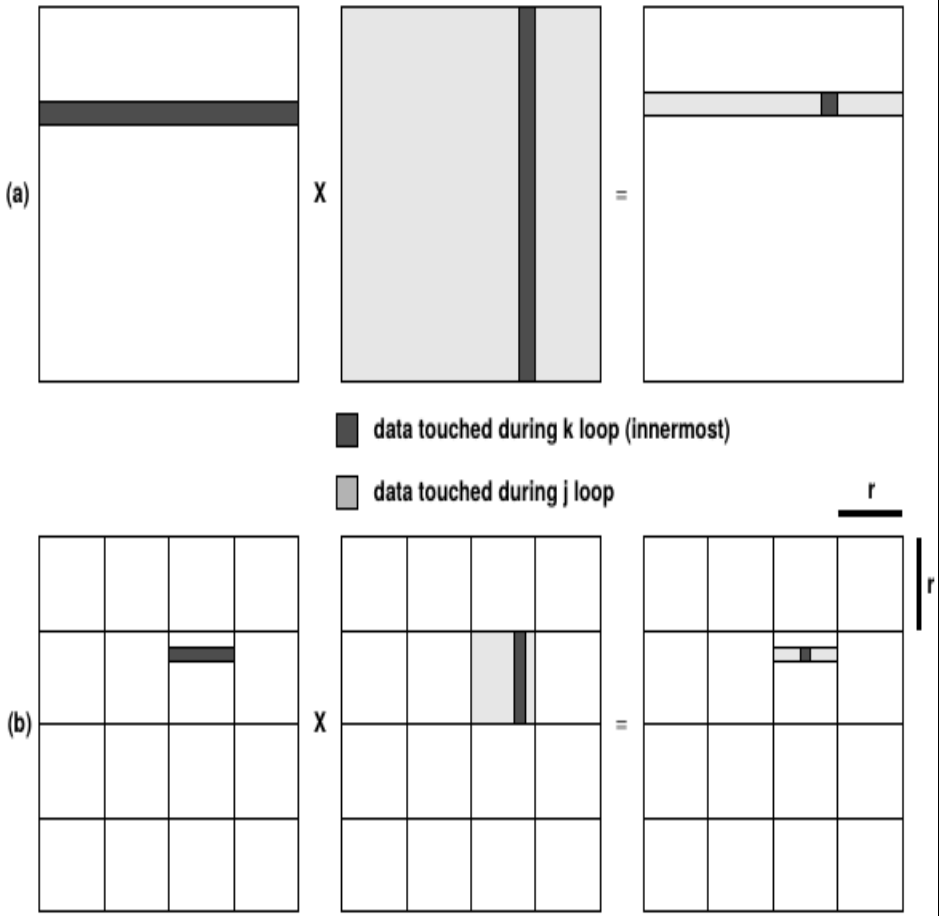

Rysunek: Stan martwy.

Mówimy, że blok znajduje się w stanie martwym między ostatnim odwołaniem, a wymianą na inny blok (wyrzuceniem z cache). Jeśli jesteśmy w stanie przewidzieć, kiedy nastąpi ten czas, możemy deaktywować daną linię (odłączyć ją od zasilania, pomysł Stefanosa Kaxirasa). Oczywiście może na tym ucierpieć wydajność, jednak według badań można ograniczyć statyczne zużycie energii o 70%.

# Proste restrukturyzacje - optymalizacje lokalności

- zamiana zagnieżdżenia pętli
- łączenie pętli
- sklejanie tablic
- cięcie na kawałki (tiling, blocking)

# Tiling (blocking)




```
for(i = 0; i < SIZE; i++)
    for(j = 0; j < SIZE; j++)
        for(k = 0; k < SIZE; k++)
            c[i][j] += a[i][k] * b[k][j];
```

```
for(jj = 0; jj < SIZE; jj += BLOCK)
    for(kk = 0; kk < SIZE; kk += BLOCK)
        for(i = 0; i < SIZE; i++)
            for(j = jj; j < jj + BLOCK; j++)
                for(k = kk; k < kk + BLOCK; k++)
                    c[i][j] += a[i][k] * b[k][j];
```

# Algorytm Mowry'iego - prefetching off-line

- algorytm reorganizujący pętle do wykorzystania w kompilatorze
- optymalizuje dostępy do tablic indeksowanych kombinacjami liniowymi iteratora pętli
- identyfikuje odwołania generujące cache-missy poprzez analize lokalności
- umieszcza instrukcje uruchamiające prefetching (tylko dla tych odwołań, które spowodowałyby cache-miss)
- wykonuje loop-unrolling (dla wykorzystania lokalności przestrzennej)
- wykonuje loop-peeling (poprzedza pierwszych PD iteracji instrukcjami prefetch)
- oblicza Prefetch Distance (PD) - ilość iteracji pętli wymaganych do przetransportowania danej do pamięci podręcznej

# Algorytm Mowry'iego - Przykład

a)
```
for (j=2; j <= N-1; j++)
  for (i=2; i <= N-1; i++)
    A[j][i]=0.25*(B[j][i-1]+B[j][i+1]+B[j-1][i]+B[j+1][i]);
```

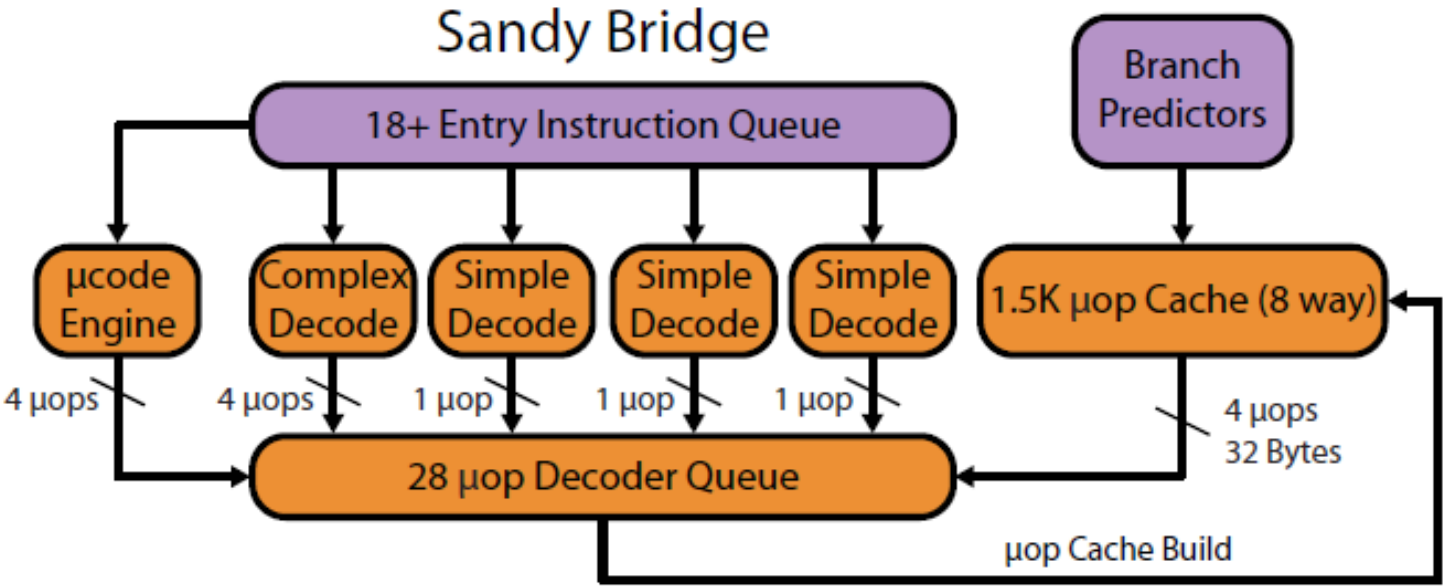
b)
```
for (j=2; j <= N-1; j++) {
  for (i=2; i <= PD; i+=4) {      //   Prologue
    prefetch(&B[j][i]);
    prefetch(&B[j-1][i]);
    prefetch(&B[j+1][i]);
    prefetch(&A[j][i]);
  }
  for (i=2; i < N-PD-1; i+=4) {   //   Steady State
    prefetch(&B[j][i+PD]);
    prefetch(&B[j-1][i+PD]);
    prefetch(&B[j+1][i+PD]);
    prefetch(&A[j][i+PD]);

    A[j][i]=0.25*(B[j][i-1]+B[j][i+1]+B[j-1][i]+B[j+1][i]);
    A[j][i+1]=0.25*(B[j][i]+B[j][i+2]+B[j-1][i+1]+B[j+1][i+1]);
    A[j][i+2]=0.25*(B[j][i+1]+B[j][i+3]+B[j-1][i+2]+B[j+1][i+2]);
    A[j][i+3]=0.25*(B[j][i+2]+B[j][i+4]+B[j-1][i+3]+B[j+1][i+3]);
  }
  for (i=N-PD; i <= N-1; i++)     //   Epilogue
    A[j][i]=0.25*(B[j][i-1]+B[j][i+1]+B[j-1][i]+B[j+1][i]);
}
```

# Uop Cache

- przechowuje zdekodowane mikroinstrukcje
- odpytywany równocześnie z instruction cache
- ogranicza wielokrotne dekodowanie tych samych instrukcji
- przyśpiesza i upraszcza pobieranie instrukcji do wykonania, przewidywanie skoków
- musi być szybki i pojemny (prawdopodobnie większy niż L1 na dane)
- prostszy niż trace cache
- wykorzystywany najnowsze procesory (Ivy Bridge, Haswell)

# Uop Cache - schemat z Sandy Bridge




Rysunek: źródło: `http://www.realworldtech.com/sandy-bridge/4/`

# Pytania

?

Wszystkie ilustracje (oprócz dotyczących Uop cache) pochodzą z *Memory systems: Cache, Dram, Disk - Bruce Jacob, Spencer Ng, David Wang*.