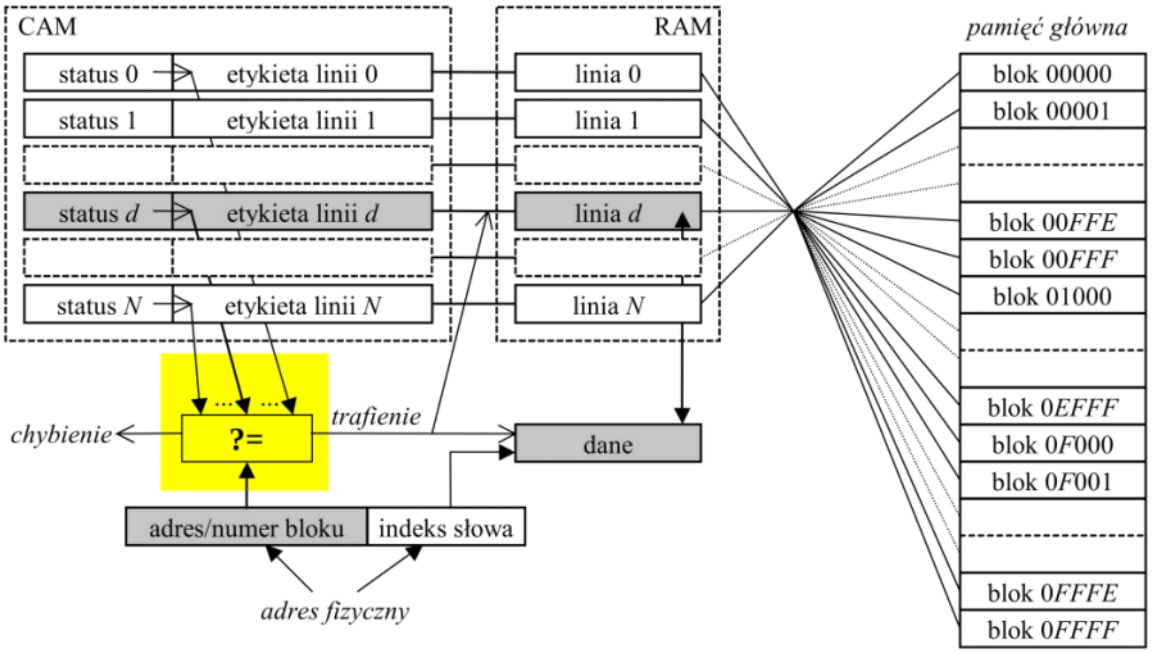
Pamięci podręczne

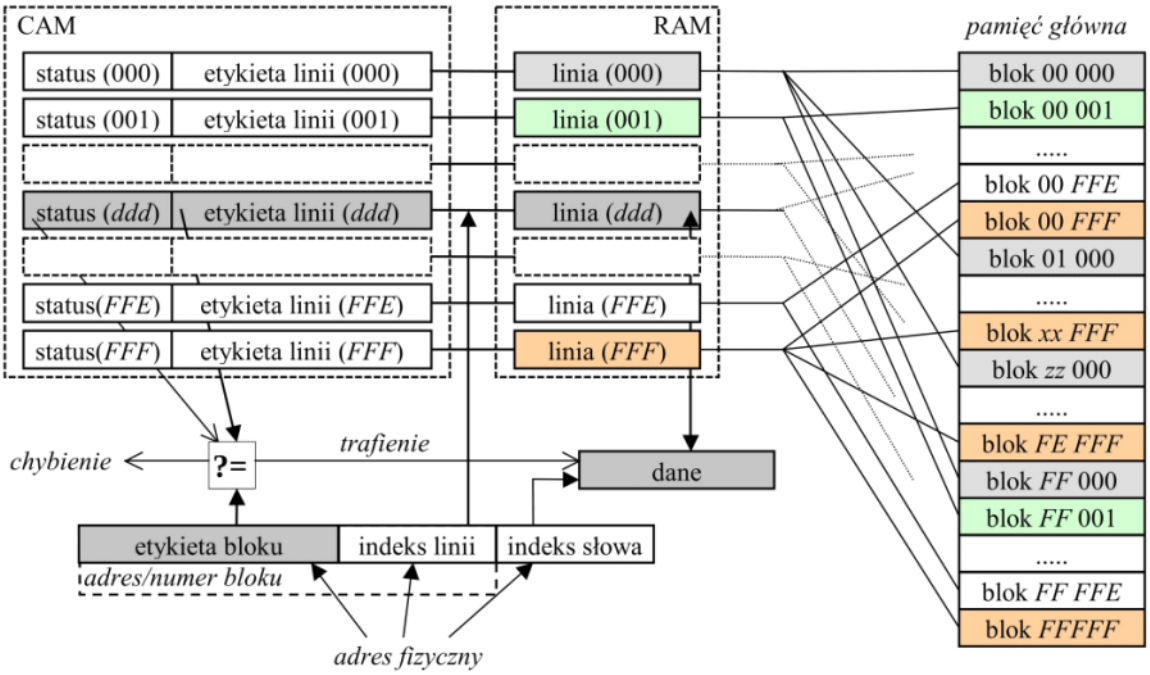
niedziela, 7 lipca 2024 22:09

Wewnątrz bufora dane przechowywane są w nieco większych blokach nazywanych liniami. W skład linii wchodzi zarówno fragment z pamięci operacyjnej jak i informacja z którego fragmentu pamięci operacyjnej są dane oraz jakieś dodatkowe bity. Linia może przechowywać dowolny fragment pamięci głównej (Pod warunkiem że adres początkowy bloku danych jest podzielny przez rozmiar linii).



Bufor całkowicie asocjacyjny (ang. *fully associative*)

W tej wersji (Odwzorowanie całkowicie skojarzone powyżej) do dowolnej linii możemy zapisać dowolny fragment pamięci operacyjnej. Jest kosztowna w realizacji i wolna bo żeby dowiedzieć się w której linii mamy kopie interesujących nas danych musimy przejść przez wszystkie linie.

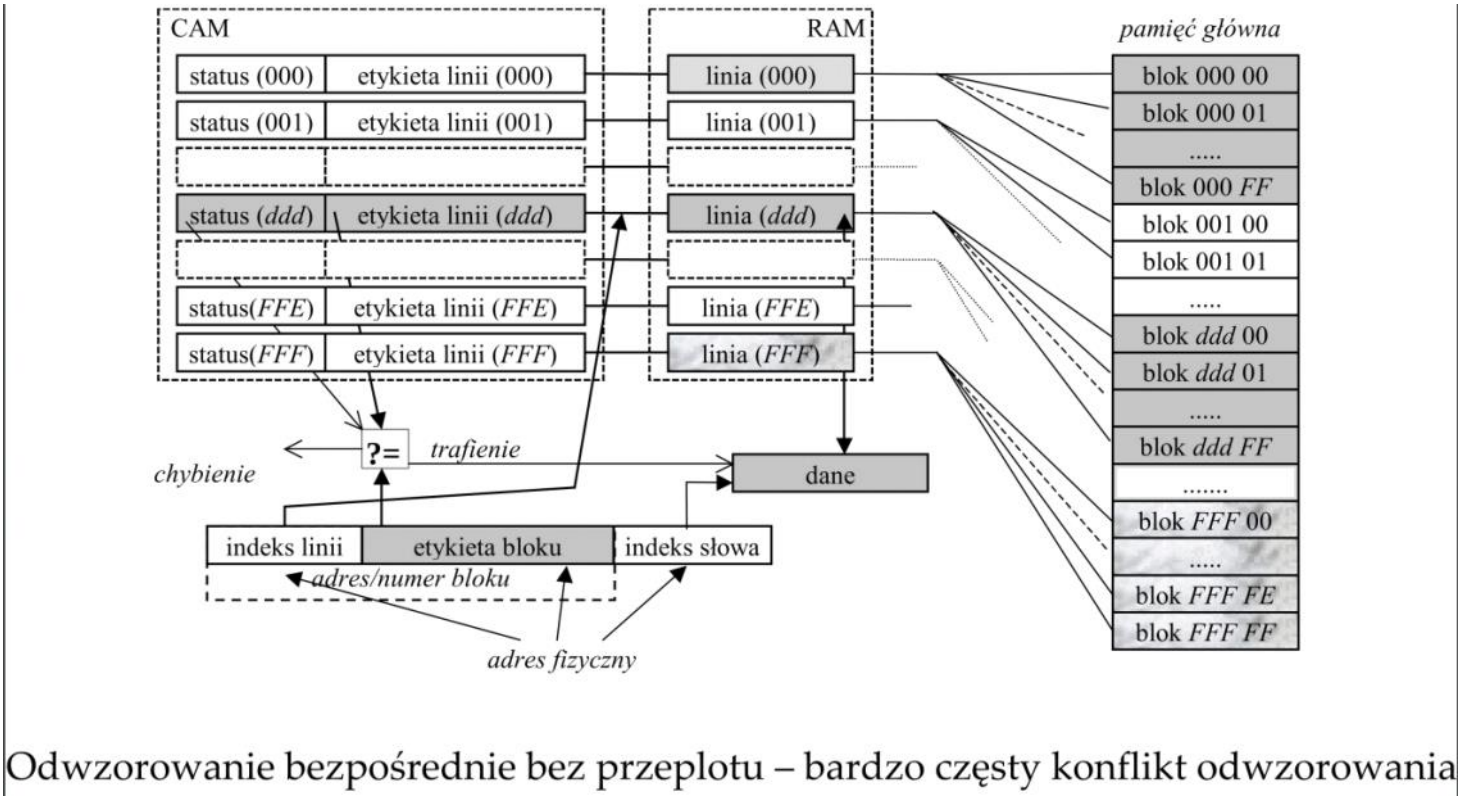


Odwzorowanie bezpośrednie (ang. *direct mapped*) z przeplotem (ang. *interlace*) bloków

Dlatego wymyślono pamięci z odwzorowaniem bezpośrednim w których do danej linii pamięci podręcznej pasują tylko niektóre linie. Które? Zależy od tego jak jest podzielony adres. Adres jest podzielony na trzy elementy

- Numer słowa w linii (To nas najmniej interesuje)
- Numer linii (Czyli kolejne bity)
- Reszta (Czyli część adresu zapisywana jako etykieta)

Etykieta jest porównywana z adresem generowanym przez procesor jeżeli są takie same to nastąpiło trafienie. Czyli dane znajdują się w pamięci podręcznej. W ten sposób przy użyciu jednego porównania wiemy czy w pamięci mamy kopie danych czy nie.



Odwzorowanie bezpośrednie bez przeplotu – bardzo częsty konflikt odwzorowania

Jako że są trzy pola to konfiguracji możemy mieć wiele ale w praktyce rozważane są tylko dwie.

- Odwzorowanie bez przeplotu. W którym wraz ze wzrostem adresu rośnie indeks słowa następnie zmieniają się etykiety a numer linii (indeks linii) pozostaje ten sam. Z tego powodu kolejny duży blok pamięci pasuje do jednej linii. Więc kiedy program odwołuje się do kolejnych danych mamy potrzebę częstej wymiany danych co powoduje dużo konfliktów. Dlatego używane jest stosunkowo rzadko.
- Odwzorowanie z przeplotem. Inaczej niż w odwzorowaniu bez przeplotu kolejne adresy pamięci pasują do różnych linii. Z tego powodu występuje znacznie mniej chybień i znacznie rzadziej musimy wymieniać dane. Jest ono też znacznie częściej używane.

W rzeczywistości pamięci podręczne są złożone z kilku, kilkunastu lub nawet kilkudziesięciu takich buforów z odwzorowaniem bezpośrednim. Takie pamięci nazywane są wielodrożnymi lub grupowo skojarzeniowymi. (Bo jest to złożenie wielu dróg a każda droga to w zasadzie prawie osobna pamięć z odwzorowaniem bezpośrednim)

Bardzo ważne terminy

niedziela, 7 lipca 2024 23:48

Kilka ważnych pojęć

Obsługa pamięci podręcznej

- unieważnianie linii (ang. *line invalidation*)
  - przed pierwszym wypełnieniem
  - wskutek zewnętrznej zmiany oryginału danych w pamięci głównej
  - przełączanie procesów – unieważnienie wszystkich linii (*line flush*)
- wypełnianie linii (ang. *line fill*) oraz wymiana linii (*line exchange*)
  - chybienie odczytu (ang. *miss on read*) lub (w trybie AOW) zapisu
- odczyt danej (ang. *read*)
  - trafienie odczytu (ang. *hit on read*)
- zapis danej (ang. *write*) → rozbieżność kopii z oryginałem
  - trafienie zapisu (ang. *hit on write*)
    - zapis skrośny (jednoczesny) (ang. *write through*, WT) – modyfikuje kopię w buforze wyższego poziomu
    - zapis lokalny (zwrotny) (ang. *write/copy back*, WB/CB) – w kopii lokalnej, opóźniony zapis do bufora wyższego poziomu podczas usuwania linii
    - zapis bezpośredni do pamięci głównej (ang. *write aside*) i unieważnienie linii
  - chybienie zapisu (ang. *miss on write*) – zapis do pamięci głównej lub bufora poziomu wyższego (NAOW) lub wypełnienie i zapis (AOW)

Do tej pory zastanawialiśmy się co się stanie jeżeli chcemy odczytywać dane z pamięci operacyjnej.

Odczyt danych

Jeżeli dane były odczytywane po raz pierwszy następuje chybienie które powoduje wypełnienie linii ale następne odwołania mogą być obsługiwane przez sterownik pamięci podręcznej który ma kopię danych.

Zapis danych

Jeżeli mamy kilka kopii tych samych danych musimy zdefiniować czy te kopie tych samych danych mogą się między sobą różnić i co zrobić jeżeli chcemy je zmodyfikować. Do tego są różne rozwiązania dwa podstawowe to:

- Zapis skrośny (write through WT). Jeżeli dane mamy zmodyfikować to te dane są zapisywane do wszystkich kopii w systemie(Do pamięci podręcznej, pamięci operacyjnej). Wadą tego rozwiązania jest czas wykonania bo każdy zapis to komunikacja z pamięcią operacyjną czyli do 100ns. Ale gwarantuje on spójność.
- Zapis lokalny (write/copy back WB/CB) Jeżeli modyfikujemy dane to zapisujemy tylko w miejscu najbliższym procesorowi. Więc w pamięci operacyjnej mamy złe dane i musimy to jakoś wychwycić i sobie z tym poradzić automatycznie lub twórca programu musi o tym pamiętać i program "jakoś tak napisać by to nie było groźne".

To wszystko pod warunkiem że dane mamy zapisane w pamięci podręcznej jeżeli ich tam nie mamy to:

- Wypełnienie i zapis (allocate on write AOW) Ten normalny zapis. Jeżeli chcemy zmodyfikować to dane najpierw kopiujemy do bufora a potem zmieniamy zgodnie z WT lub WB/CB. Tak działa większość maszyn.
- Zapis bezpośredni do pamięci głównej lub do bufora poziomu wyższego (not allocate on write NAOW) Nie ma wcześniejszego zapisu jak w AOW

```
 7 void copy_simple (const V_AVX512 * src, V_AVX512 * dst, size_t nVectors)
 8 {
 9   for (size_t i=0 ; i < nVectors ; i++)
10   {
11     dst [i] = src [i] ;
12   }
13 }
```

```
16,69 | 10: →vmovaps    (%rdi,%rax,1),%zmm0
33,90 |      vmovaps    %zmm0,(%rsi,%rax,1)
17,52 |      add        $0x40,%rax
31,82 |      dec        %rdx
 0,08 |      jne        10
```

2 odczyty, jeden zapis
8 GiB / 0.25 s = **34 GB/s**

```
15 void copy_nt (const V_AVX512 * src, V_AVX512 * dst, size_t nVectors)
16 {
17   for (size_t i=0 ; i < nVectors ; i++)
18   {
19     V_AVX512 v = src [i] ;
20     __builtin_nontemporal_store (v, dst+i) ;
21   }
```

```
16,24 | 10: →vmovaps    (%rdi,%rax,1),%zmm0
34,45 |      vmovntps   %zmm0,(%rsi,%rax,1)
17,28 |      add        $0x40,%rax
32,03 |      dec        %rdx
      |      jne        10
```

Jeden odczyt, jeden zapis
8 GiB / 0.167 s = **51 GB/s**

51 / 34 = 1.5

To jest prawdziwe pod warunkiem że przeczytaliśmy dokładnie 8Gb bo wtedy po lewej zostało "tak jakby" przesłane zostało 24Gb (8Gb odczytane+ 8Gb zapisane w pamięci podręcznej + 8Gb zapisane w pamięci operacyjnej) a po prawej tylko 16Gb (8Gb odczytane + 8Gb zapisane w drugiej tablicy). Stąd 1.5 razy szybciej . Komputer normalnie działa w trybie AOW ale można wymusić NAOW przy użyciu specjalnych instrukcji (po prawej).

No i teraz tak yhhhhh

poniedziałek, 8 lipca 2024 18:02

To jest ważne (pokazuje na coś w prezentacji związanego z modelami spójności pamięci nie znalazłem tego). Jeżeli stosujemy buforowanie zapisów top w różnych procesorach różne modele spójności pamięci mogą zmienić kolejność wykonywania działań na pamięci. Więc w skrajnych przypadkach może być źle(np. Najpierw rozkaz zapisu potem rozkaz odczytu daje wyniki "na odwrót albo jeszcze coś dziwnego"). Te problemy występują niezależnie od użytego języka. Według Tomczaka tak jest szybciej ale programista musi o tym pamiętać.

Table 7-2. Memory Access by Memory Type

Memory Access Allowed		Memory Type				
		UC/CD	WC	WP	WT	WB
Read	Out-of-Order	no	yes	yes	yes	yes
	Speculative	no	yes	yes	yes	yes
	Reorder Before Write	no	yes	yes	yes	yes
Write	Out-of-Order	no	yes	no	no	no
	Speculative	no	no	no	no	no
	Buffering	no	yes	yes	yes	yes
	Combining[1]	no	yes	no	yes	yes

Note:

1. *Write-combining buffers are separate from write (store) buffers.*

Table 7-3. Caching Policy by Memory Type

Caching Policy	Memory Type					
	UC	CD	WC	WP	WT	WB
Read Cacheable	no	no	no	yes	yes	yes
Write Cacheable	no	no	no	no	yes	yes
Read Allocate	no	no	no	yes	yes	yes
Write Allocate	no	no	no	no	no	yes
Write Hits Update Memory	yes[2]	yes[1]	yes[2]	yes[3]	yes	no

Note:

1. *For the L1 data cache and the L2 cache, if an access hits the cache, the cache line is invalidated. If the cache line is in the modified state, the line is written to main memory and then invalidated. For the L1 instruction cache, read (instruction fetch) hits access the cache rather than main memory.*
2. *The data is not cached, so a cache write hit cannot occur. However, memory is updated.*
3. *Write hits update memory and invalidate the cache line.*

33

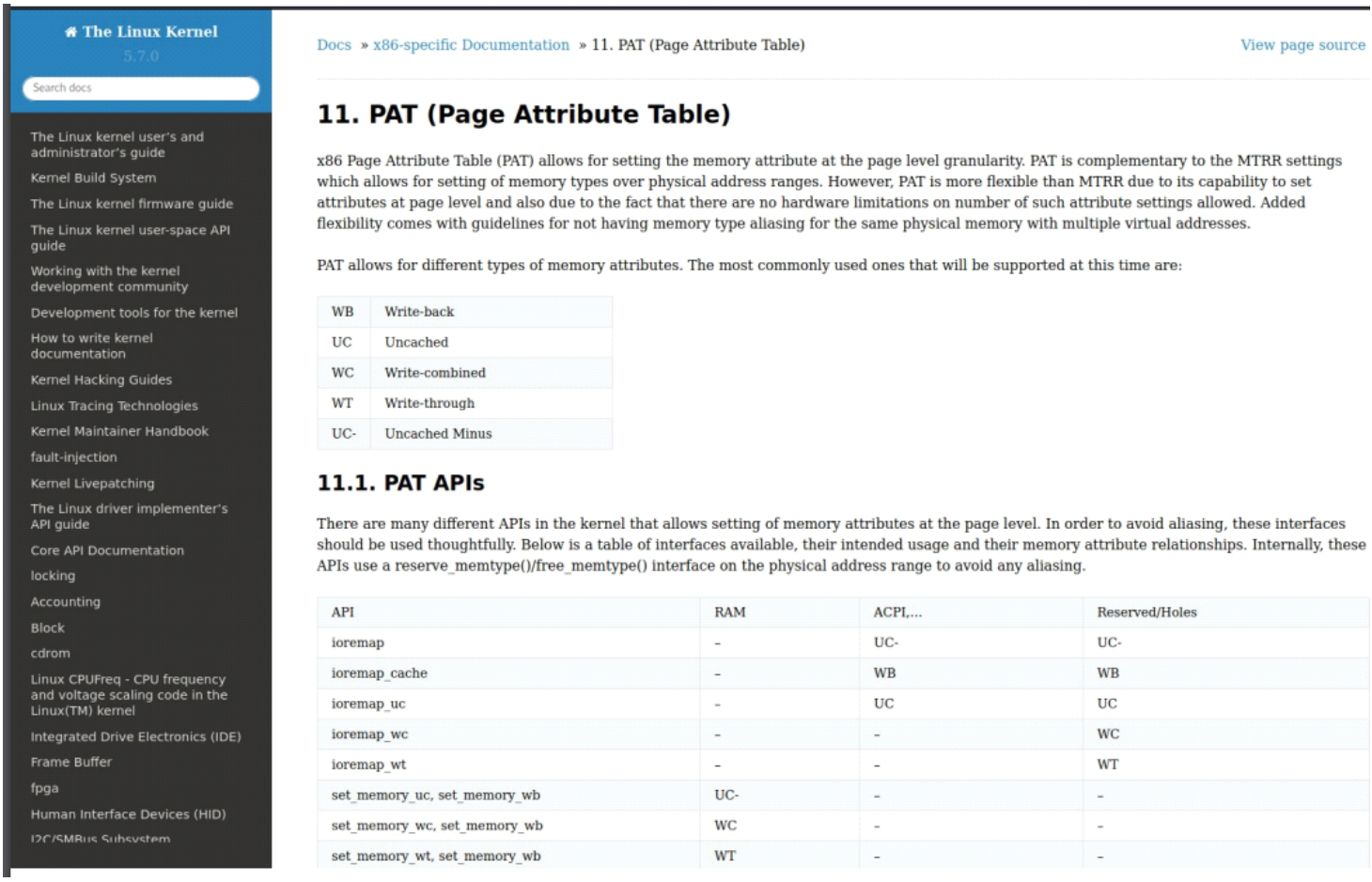
Typy zapisu

- Skrośny WB
- Zwrotny WT
- Zabezpieczony WP
- Kombinowany WC
- "Taki taki" CD
- Un cached UC (Pamięć nie obsługiwana przez bufor. Tam mogą być zmapowane urządzenia wejścia/wyjścia)

Co zmieniają te tryby pracy:

- Kolejność wykonywania operacji
- Pobieranie na zapas (spekulatywne)
- Kolejność operacji PRZED zapisem

Procesory mają te opcje różnie po ustawiane więc można je sobie (w pewnym zakresie) ręcznie pozmieniać jeżeli chcemy "Świadomie i poprawnie" korzystać z pamięci



The Linux Kernel
5.7.0

Search docs

- The Linux kernel user's and administrator's guide
- Kernel Build System
- The Linux kernel firmware guide
- The Linux kernel user-space API guide
- Working with the kernel development community
- Development tools for the kernel
- How to write kernel documentation
- Kernel Hacking Guides
- Linux Tracing Technologies
- Kernel Maintainer Handbook
- fault-injection
- Kernel Livepatching
- The Linux driver implementer's API guide
- Core API Documentation
- locking
- Accounting
- Block
- cdrom
- Linux CPUFreq - CPU frequency and voltage scaling code in the Linux(TM) kernel
- Integrated Drive Electronics (IDE)
- Frame Buffer
- fpga
- Human Interface Devices (HID)
- I2C/SMBus Subsystem

Docs » x86-specific Documentation » 11. PAT (Page Attribute Table) View page source

11. PAT (Page Attribute Table)

x86 Page Attribute Table (PAT) allows for setting the memory attribute at the page level granularity. PAT is complementary to the MTRR settings which allows for setting of memory types over physical address ranges. However, PAT is more flexible than MTRR due to its capability to set attributes at page level and also due to the fact that there are no hardware limitations on number of such attribute settings allowed. Added flexibility comes with guidelines for not having memory type aliasing for the same physical memory with multiple virtual addresses.

PAT allows for different types of memory attributes. The most commonly used ones that will be supported at this time are:

WB	Write-back
UC	Uncached
WC	Write-combined
WT	Write-through
UC-	Uncached Minus

11.1. PAT APIs

There are many different APIs in the kernel that allows setting of memory attributes at the page level. In order to avoid aliasing, these interfaces should be used thoughtfully. Below is a table of interfaces available, their intended usage and their memory attribute relationships. Internally, these APIs use a reserve_memtype()/free_memtype() interface on the physical address range to avoid any aliasing.

API	RAM	ACPI,...	Reserved/Holes
ioremap	–	UC-	UC-
ioremap_cache	–	WB	WB
ioremap_uc	–	UC	UC
ioremap_wc	–	–	WC
ioremap_wt	–	–	WT
set_memory_uc, set_memory_wb	UC-	–	–
set_memory_wc, set_memory_wb	WC	–	–
set_memory_wt, set_memory_wb	WT	–	–

Np. w linuxsie jest cały zestaw pamięci pozwalający zmieniać te tryby
Pada pytanie czy nie lepiej po prostu napisać program a potem zmieniać tryby zapisu żeby zobaczyć w którym jest najszybciej?
Tomczak stwierdza że nie bo problemy które to może spowodować występują "Co 38 godziny jeżeli proces uruchamiany jest na 7 rdzeniach i jest 29 stycznia" (Względnie mega rzadko i w specyficznych warunkach). Czyli są to zachowania niedeterministyczne (Jest tyle zmiennych że jedna zmiana może powodować całkowitą zmianę wyniku, Efekt Motyla)

Trochę konkretów

poniedziałek, 8 lipca 2024 18:28

Kiedy korzystamy z pamięci chcemy minimalizować liczbę chybień. I jak najczęściej mieć trafienia

Przyczyny braku trafienia w buforze *cache* można zakwalifikować do 3 kategorii (ang. *three C's model: compulsory-capacity-conflict*):

– nieuniknione (ang. *compulsory*):
pierwsza próba dostępu do bloku musi skutkować chybieniem, bo blok nie może być odwzorowany w buforze – nie było wcześniej zapotrzebowania;

– (ograniczona) pojemność (ang. *capacity*):
jeśli bufor jest zbyt mały, aby pomieścić wszystkie bloki potrzebne podczas wykonania programu, niektóre bloki muszą być usuwane z bufora i później ewentualnie ponownie kopiowane;

– konflikt (ang. *conflict*):
jeśli odwzorowanie bloków nie jest w pełni skojarzeniowe (ang. *fully associative*), odwzorowanie bloku w grupie może wymagać usunięcia innego bloku, mimo wolnych miejsc w innych grupach;

w systemie wieloprocesorowym także:

– spójność (ang. *coherency*) – brak bloku w buforze może być spowodowany działaniem innego procesora, używającego tej samej pamięci operacyjnej.

(Bardzo ważne)
Model 3 C. Chybienia mogą być:

- Nieuniknione (Compulsory) Kiedy chcemy odczytać dane ale nie mamy ich w pamięci podręcznej (No chyba że jest stosowane pobieranie poprzedzające ale o tym zaraz). I na to prawie nic nie poradzimy.
- Pojemność (Capacity) Może być tak że mamy zbyt małą pojemność pamięć podręcznej względem zbioru danych. Tu wystarczy tak napisać program by lokalność przestrzenna była dla mniejszego zbioru danych (Zamiast przetwarzać całą macierz dzielić ją na mniejsze "kwadraciki" które mieszczą się w pamięci podręcznej)
- Konflikt (Conflict) W przypadku pamięci blokowo skojarzeniowych i wielodrożnych występują konflikty. Czyli wiele różnych adresów z pamięci operacyjnej pasuje do jednej linii. Można tak napisać kod by zawsze odwoływać się do elementów pamięci pasujących do różnych linii.

No i jeszcze 4 kategoria która się w trzech nie mieści ale też jest na C

- Spójność (Coherency) Jeżeli mamy więcej niż jedno urządzenie operujące na pamięci może się tak zdarzyć że jakieś urządzenie zmieni dane w pamięci a my w pamięci podręcznej mamy dane nie aktualne. I wtedy może (ale nie musi) być chybienie (Zależnie od modelu spójności)

1. Zwiększenie rozmiaru bloku/linii
   – czynnik/efekt/zjawisko: wzrost lokalności przestrzennej
   * wady: wzrost strat czasu w razie chybienia (dłuższy czas kopiowania bloku)
2. Zwiększenie pojemności bufora cache
   – czynnik/efekt/zjawisko: słabszy efekt ograniczonej pojemności
   * wady: dłuższy czas dostępu, większy pobór energii, wyższy koszt
3. Lepszy efekt skojarzeniowości
   – czynnik/efekt/zjawisko: rzadsze chybienia wskutek konfliktu
   * wady: dłuższy czas wyszukiwania bloku i w efekcie czas dostępu
4. Bufor wielopoziomowy
   – czynnik/efekt/zjawisko: mały szybki bufor 1. poziomu, duży, wolniejszy bufor 2.poz.
   * wady: trudniejsza obsługa w porównaniu z buforem jednopoziomowym
5. Priorytet chybień odczytu nad zapisem
   – czynnik/efekt/zjawisko: - dodatkowy bufor zapisu
   * wady: możliwy hazard odczyt po zapisie (ang. *read after write*)
6. Unikanie translacji adresu wirtualnego
   – czynnik/efekt/zjawisko: identyfikatorem w cache jest adres rzeczywisty
   * wady: rozmiar bufora 1.pozimu równy rozmiarowi strony

Co zrobić żeby jak najrzadziej chybiać

1. Zwiększyć rozmiar linii. Jeżeli rozmiar zwiększamy to możemy dojść do momentu gdy linia ma 1Kb a my potrzebujemy tylko 12b. Więc każde chybienie powoduje przesłanie 1Kb tylko po to by użyć 12b. Więc zazwyczaj rozmiar linii pasuje do rozmiaru transakcji. Obecnie od 32b do 64b (czasami 128b może nawet 256b ale Tomczak nie wie czy takie naprawdę istnieją)
2. Zwiększenie bufora. Ale im większa pamięć tym wolniejsza. Jeżeli by tak nie było to bufory były by zbędne
3. Stosuje się więcej dróg (Często stosowane). Jak mamy więcej dróg to mamy więcej linii pasujących do tych samych fragmentów pamięci operacyjnej i jest mniejsza szansa na konflikt.
4. Zastosowanie bufora wielopoziomowego. Trochę rozwinięcie punktu 2. Tyle że stosuje wiele coraz większych (i wolniejszych buforów). Obecnie najbliżej procesora mamy bufory rzędu dziesiątek kilo-bajtów. Trochę dalej setki kilo-bajtów potem mega-bajty itd.
5. Oszczędzanie pamięci podręcznej. Czyli rozwiązania podobne do NAOW gdzie nie odwołania do pamięci nie powodują wykorzystania linii. Jeżeli wiemy że danych nie będziemy używać wielokrotnie to można ich nie zapisywać do pamięci podręcznej
6. "Taki myk w sprzęcie". Sprawia że pamięć podręczna będzie działać szybciej. Tym się nie będziemy zajmować

Poza tym

poniedziałek, 8 lipca 2024 19:23

Żeby efektywnie używać pamięci podręcznej trzeba wiedzieć jak adresy do których się odwołujemy powodują zajmowanie poszczególnych linii w pamięci podręcznej.
Są trzy (podstawowe) strategie

- Wypełniania pamięci podręcznej (O tym było wspominane w poprzednich sekcjach)
- Wybierania którą linię z pamięci podręcznej usunąć. Występuje gdy pamięć skojarzeniowa się zapełni lub gdy wszystkie drogi się zapełnią. Wtedy trzeba wybrać którą linię usunąć lub z której drogi linię usunąć. Wymiana może być:
  - Losowa - da średnie wyniki ale raczej rzadko używana
  - Linia która będzie najmniej lub najpóźniej potrzebna (według naszych przewidywań). Jak że zazwyczaj nie wiemy co będzie w przyszłości to zwykle stosowany jest algorytm LRU (last recently used). Z każdą linią kojarzymy licznik który mówi kiedy ta linia była ostatnio używana. Przy wymianie sprawdzane są te liczniki i wyrzucana jest ta linia której dawno nie używaliśmy(ma największy licznik)
- Kiedy wypełniać linie pamięci podręcznej. Występują dwa rodzaje
  - Wymuszone. Do tej pory mówiliśmy o tym. Wypełnianie tylko wtedy gdy odwołujemy się do danych. Czasochłonne.
  - Uprzedzające. Można przewidywać jakie dane będą potrzebne i pobierać je wcześniej ograniczając stratę czasu.

Kiedy proces jest zatrzymywany tylko część pamięci podręcznej (Związana wyłącznie z procesem) jest usuwane (Np. Jeżeli proces korzystał z biblioteki języka C to kopie kodów funkcji z języka C nie są usuwane). System operacyjny zajmuje się usuwaniem odpowiednich części pamięci

probationary segment, giving this line another chance to be accessed before being replaced. The size limit of the protected segment is an SLRU parameter which varies according to I/O workload patterns. When data must be discarded from the cache, lines are obtained from the LRU end of the probationary segment.[10]

LRU approximations [edit]

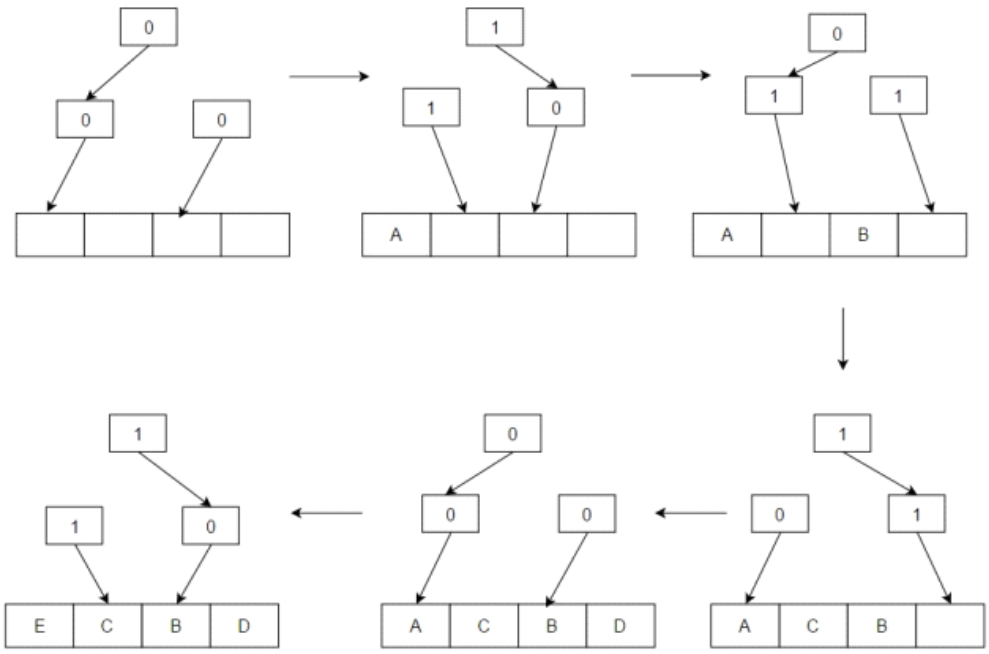
LRU may be expensive in caches with higher associativity. Practical hardware usually employs an approximation to achieve similar performance at a lower hardware cost.

Pseudo-LRU (PLRU) [edit]

Further information: Pseudo-LRU

For CPU caches with large associativity (generally > four ways), the implementation cost of LRU becomes prohibitive. In many CPU caches, an algorithm which almost always discards one of the least-recently-used items is sufficient; many CPU designers choose a PLRU algorithm, which only needs one bit per cache item to work. PLRU typically has a slightly-worse miss ratio, slightly-better latency, uses slightly less power than LRU, and has a lower overhead than LRU.

Bits work as a binary tree of one-bit pointers which point to a less-recently-used sub-tree. Following the pointer chain to the leaf node identifies the replacement candidate. With an access, all pointers in the chain from the accessed way's leaf node to the root node are set to point to a sub-tree which does not contain the accessed path. The access sequence in the example is A B C D E:



When there is access to a value (such as A) and it is not in the cache, it is loaded from memory and placed in the block where the arrows are pointing in the example. After that block is placed, the arrows are flipped to point the opposite way. A, B, C and D are placed; E replaces A as the cache fills because that was where the arrows were pointing, and the arrows which led to A flip to point in the opposite direction (to B, the block which will be replaced on the next cache miss).

Schemat działania algorytmu LRU (Pojedyncze komórki nad paskiem to przełączniki kontrolujące linie). Kiedy zapisujemy odwracamy wartość w przełączniku co kieruje nas do innej linii dzięki temu nie musimy przechowywać z każdą linią licznika a wystarczy prosty algorytm

Kiedy pobierać linie

poniedziałek, 8 lipca 2024 19:54

Linia jest pobierana gdy chcemy użyć danych a nie mamy ich w pamięci podręcznej. Jednakże to powoduje przerwanie kilkadziesiąt czy 100 ns. Czyli chybienie będzie nieuniknione. Istnieją jednak metody pobierania wyprzedzającego. Jest to możliwe ponieważ po przesłaniu magistrala "siedzi" i czeka na następny przesył więc czemu z tego nie skorzystać.

Najprostszym sposobem przewidywania jest pobieranie lini obecnej i następnej (i,i+1). Bo ta następna może się przydać
Ten sposób można jeszcze podzielić na trzy mniejsze pod sposoby

- Pobieramy zawsze wtedy gdy odwołujemy się do danych
- Pobieramy tylko wtedy gdy pobieraliśmy linię wcześniejszą
- Pobieramy dwie linie i drugą zaznaczamy żeby kolejnych już nie pobierać 45:48

Wykład do połowy bo mnie szlak trafił XD