Model pamięci w systemach wieloprocesorowych

Paweł Dziepak

Instytut Informatyki Uniwersytetu Wrocławskiego

18 listopada 2014

UMA vs. NUMA

Uniform memory access (UMA)

Czas dostępu do dowolnego obszaru pamięci jest taki sam dla wszystkich procesorów. W najprostszej implementacji wszystkie procesory są połączone wspólną szyną do pamięci. Bardziej skomplikowane rowzwiązania wykorzystują np. *crossbar switch*.

Non-uniform memory access (NUMA)

Dla każdego procesora pewne obszary pamięci są lokalne i dostęp do nich jest szybki, podczas gdy dostęp do pozostałych trwa znacząco dłużej.

Problemy w systemach wieloprocesorowych

- propagacja zapisów i spójność cache (*cache coherence*)
- dostęp do szyny systemowej
- operacje atomowe
- obserwowalna kolejność dostępów do pamięci

Protokoły spójności pamięci

Każdej lini pamięci podręcznej przypisane są dodatkowe bity oznaczające jej aktualny stan. Na podstawie wykonywanych instrukcji oraz działań innych procesorów stan ten jest odpowiednio aktualizowany.

Protokół MESI

- **Modified** – linia cache nie jest obecna w pamięci podręcznej żadnego z pozostałych procesorów, ponadto zawarte w niej dane zostały zmienione a pamięć RAM nie została jeszcze zaktualizowana
- **Exclusive** – linia cache nie jest obecna w pamięci podręcznej żadnego z pozostałych procesorów, zawarte w niej dane są zgodne z zawartością pamięci RAM
- **Shared** – linia cache znajduje sie w pamięciach podręcznych wielu procesorów, jej zawartość jest zgodna z zawartością pamięci RAM
- **Invalid** – linia cache nie zawiera poprawnych danych

Protokół MOESI

Protokół MOESI jest modyfikacją protokołu MESI w której dodano dodatkowy stan **Owned** i zmodyfikowano stan **Shared**.

- **Owned** – linia cache znajduje sie w pamięciach podręcznych wielu procesorów, zawarte w niej dane zostały zmienione a pamięć RAM nie została jeszcze zaktualizowana, tylko jeden procesor może mieć daną linię cache w tym stanie.
- **Shared** – linia cache znajduje sie w pamięciach podręcznych wielu procesorów, jej zawartośc jest zgodna z zawartością pamięci RAM $\iff$ żaden inny procesor nie posiada tej lini w stanie **Owned**.

Dodanie stanu **Owned** pozwala na współdzielenie danych między procesorami bez konieczności wykonywania zapisów do pamięci RAM.

Protokół MESIF

Protokół MESIF został zaprojektowany z myślą o architekturze ccNUMA. Występuje dodatkowy stan **Forward**.

- **Forward** – jak w przypadku **Shared** z tym, że tylko procesor posiadający linię cache w tym stanie odpowiada na żądania jej odczytu wysłane przez inne procesory.

Linię cache w stanie **Forward** posiada procesor który jako ostani pobrał ją do swojej pamięci podręcznej.

Sposoby implementacji protokołów spójności

Jak poprawnie aktualizować stany poszczególnych lini cache?

- **Snooping** – wszystkie procesory na bieżąco śledzą cały ruch na szynie systemowej i w zależności od pojawiających się żądań wykonują odpowiednie przejścia między stanami implementowanego protokołu.
- **Directory-based** – istnieje wspólny słównik który zawiera informacje o stanie danej lini cache oraz o procesorach które posiadają ją w swojej pamięci cache.

Dostęp do szyny systemowej

Zwykle wykorzystywana jest technika *split transactions*. Procesor uzyskuje dostęp do szyny systemowej, wysyła żądanie, a następnie ją zwalnia i oczekuje na odpowiedź. *Miss-status handling registers* przechowuje informacje o wszystkich niezakończonych transakcjach. MSHR pozwala także na połączenie kolejnych dostępów do tych samych adresów pamięci.

W *write-back buffer* znajdują się linie cache które usunięto z pamięci podręcznej procesora, ale nie zostały jeszcze zapisane do pamięci.

Dostęp do szyny systemowej, cd.

Fill buffer zbiera dane z transakcji aż odczytana zostanie cała linia cache.

Do *snoop queue* trafiają żądania wysłane przez inne procesory które muszą być przetworzone przez protokół spójności cache.

Instrukcje atomowe

Współdzielenie pamięci przez wiele procesorów często wymaga wykonywania operacji niepodzielnych (w celu implementacji algorytmów *lock-free* bądź prymitywów synchronizacyjnych). Poszczególne architektury sprzętowe oferują różne sposoby wykonywania takich instrukcji:

- wyrównanie zapisy i odczyty do pamięci o rozmiarze do 64 bitów w przypadku wielu architektur zawsze są atomowe
- możliwość zablokowania szyny systemowej na czas wykonywania instrukcji (np. x86)
- wykorzystanie protokołu spójności cache do upewnienia się, że instrukcja została wykonana niepodzielnie (np. aarch64)

x86: prefiks `lock`

W przypadku x86 niektóre operacje mogą być poprzedzone prefiksem `lock` który powoduje zablokowanie szyny systemowej podczas wykonywania tej instrukcji co gwarantuje atomowość operacji *read-modify-write*, np.:

```
lock xadd $1, (%rax)
```

Niektóre czynności zawsze wykonywane są z zablokowaną szyną, np. ustawianie bitów *accessed* i *dirt* w katalogu stron.

aarch64: link-load/store-conditional

W schemacie LL/SC, zapis zostanie wykonany tylko jeśli żaden inny procsor nie zmodyfikował odczytanej wcześniej wartości. Zwykle monitorowany jest większy obszar pamięci niż wymagany co powoduje, że należy szczególnie uważać na *false sharing*.

```
.1:
    ldxr x1, [x0]
    add  x1, x1, 1
    stxr w2, x1, [x0]
    cbnz w2, .1b
```

Spójność pamięci a *out-of-order execution*

Wątek 1 (producent)

```
ring[p] = obj;
p++;
```

Wątek 2 (konsument)

```
if (c < p) {
    return ring[c++];
}
return nullptr;
```

Modele spójności pamięci

- **sequential consistency** – wszystkie operacje stają się widoczne dla pozostałych procesorów bez zmiany kolejności
- **total store order** (x64, sparcv9) – zapisy (a także instrukcje atomowe) są widoczne w takiej kolejności w jakiej są zapisane w kodzie programu
- **relaxed memory order** (aarch64, IA-64) – obserwowalna kolejność zależnych zapisów nie ulega zmianie
- **DEC Alpha** (DEC Alpha) – brak jakichkolwiek gwarancji

Modele spójności pamięci – Alpha

Wartości początkowe

```
v0 = 1; v = 0; p = &v0;
```

Wątek 1

```
v = 1; mfence(); p = &v;
```

Wątek 2

```
i = *p;
```

Możliwym rezultatem jest `i = 0`.

Bariery pamięci

Pełna bariera pamięci wymusza aby wszystkie operacje dostępu pamięci znajdujące sie w kodzie programu przed nią stały się widoczne przed wykonaniem jakiegokolwiek dostępu do pamieci znajdującego się po tej barierze.

Często ISA oferuje także słabsze bariery które np. wymuszają porządek jedynie operacji zapisu (np. `sfence` w x86). Mogą także istnieć inne instrukcje które także wymuszają określony porządek (np. operacja odczytu `ldar` w aarch64).

Wysokopoziomowe spojrzenie na bariery pamięci

- `std::memory_order_relaxed` – kolejność instrukcji nie jest istotna
- `std::memory_order_consume` – zapisy do zależnych adresów pamieci wykonane przez inne wątki stają się widoczne
- `std::memory_order_acquire` – zapisy wykonane przez inne wątki stają się widoczne
- `std::memory_order_release` – zapisy wykonane przez ten wątek stają się widoczne dla innych
- `std::memory_order_acq_rel` – `acquire` i `release`
- `std::memory_order_seq_cst` – kolejność działań jak w kodzie programu

Instrukcje `acquire` lub `consume` zwykle powinny być sparowane z instrukcją `relaxed`.

`std::memory_order` i x64

x64 oprócz gwarantownia TSO nie zmienia także kolejności odczytów względem siebie (może natomiast wykonać odczyt przed zapisem).

- `store seq_cst` – zapis z użyciem atomowej instrukcji `xchg`
- `seq_cst fence` – pełna bariera pamięci `mfence`

Pozostałe instrukcje bez dodatkowych zmian.

`std::memory_order` i aarch64

- `load acquire` – `ldar` = odczyt `ldr` + gwarancja wykonania przed kolejnymi zapisami i odczytami
- `store release` – `stlr` = zapis `str` + gwarancja wykonania po poprzedzających zapisch i odczytach
- `acquire fence` – bariera pamięci `dmb ld` – tylko odczyty
- `seq_cst fence` – pełna bariera pamięci `dmb`