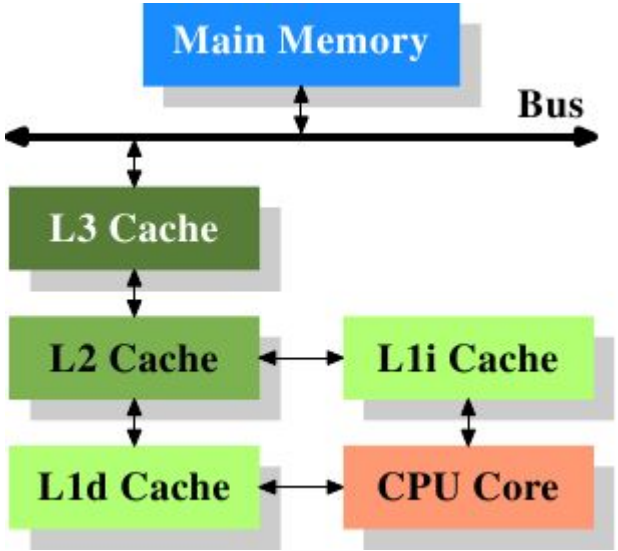
Операционные системы

Кеши и мультипроцессирование

Кеш

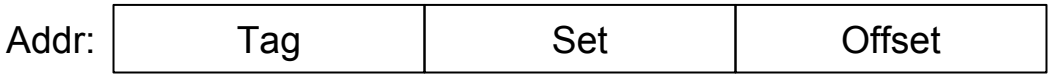


Ускорение доступа к оперативной памяти.

Обычно многоуровневый.

В многоядерных системах часть уровней общие, часть индивидуальные

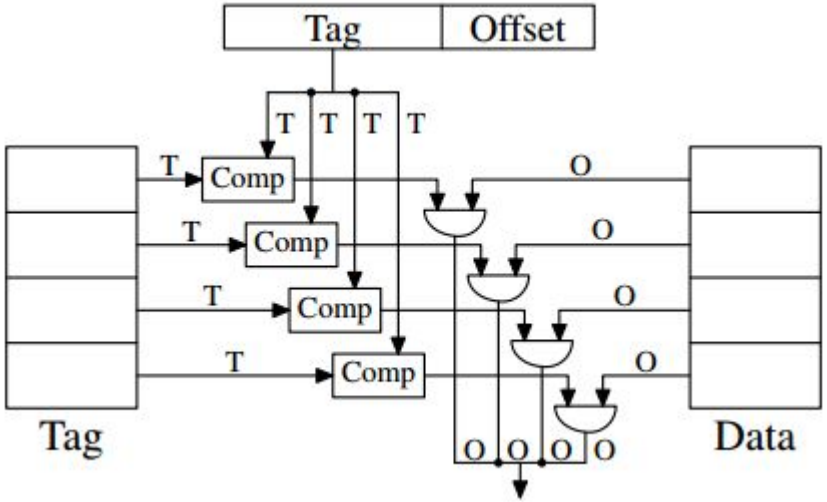
Устройство кэша



- 2^len(offset) - длина линейки
- 2^len(set) - количество наборов
- ассоциативность - количество tag в одном наборе
- Частные случаи:
  - полноассоциативный
  - с прямым отображением

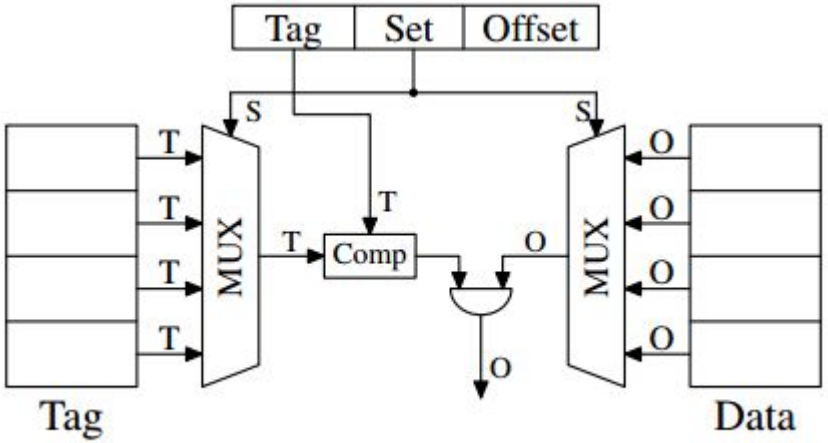
Полноассоциативный

len(set) == 0

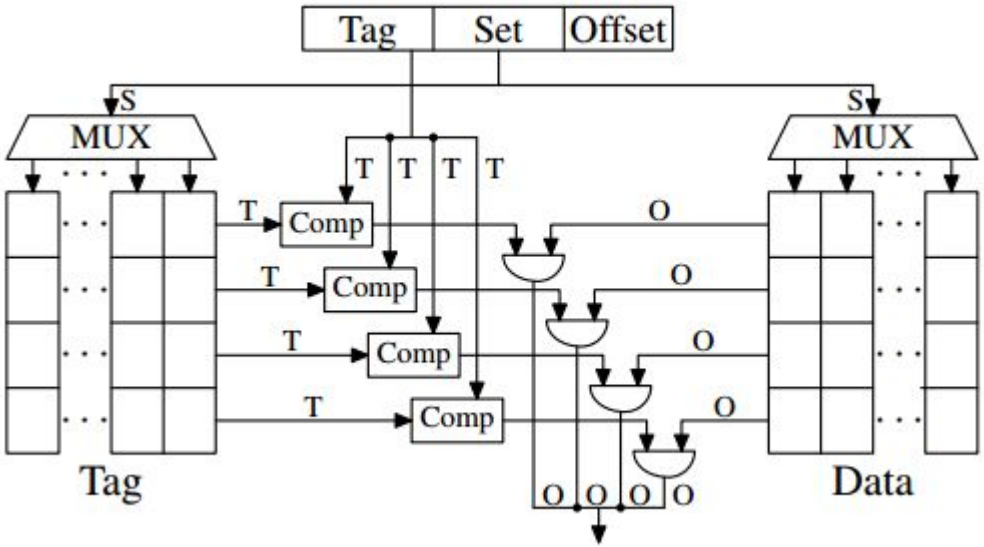


С прямым отображением

1-ассоциативный кеш



Ассоциативный кэш



Запись в память

Политики работы кэша

- write-through: запись производится и в кэш и в память
- write-back: отложить запись до выгрузки линейки
  - или до момента освобождения шины

Проблемы мультипроцессорах

Рассматриваем, например, язык C:

a = 1;

b = 2;

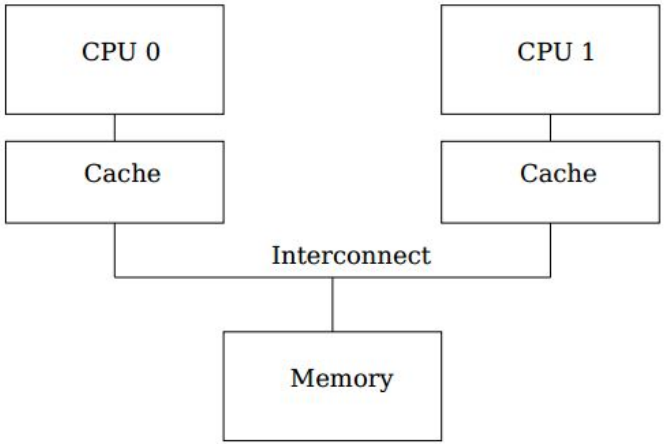
Выражения независимы, компилятор имеет право переставить их. Должно быть

a = 1;

__asm__ __volatile__ (: : : “memory”);

b = 2;

Кэши и мультипроцессоры



1. CPU0 пишет 0x1000
2. CPU1 читатет 0x1000 и не видит изменений

Paul E. McKenney. Memory Barriers: a Hardware View for Software Hackers

Протокол поддержания корректности кэшей

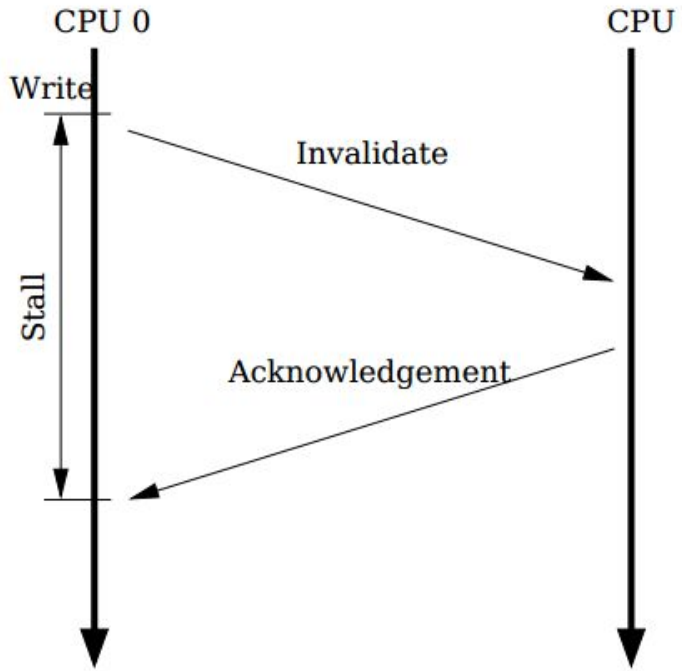
MESI:

- Modified
- Exclusive
- Shared
- Invalid

Сообщения

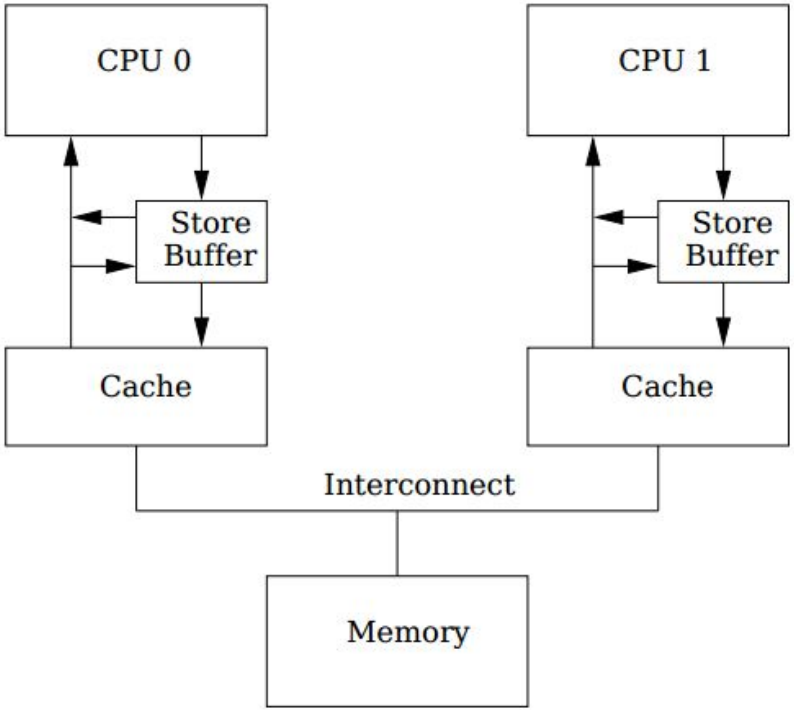
- Read
- Read Response
- Invalidate
- Invalidate Ack
- Read Invalidate
- Writeback

MESI пример



CPU0 хочет записать данные, поэтому делает Read Invalidate и ожидает, пока не придёт Ack

Буферы записи



Поэтому, записи помещаются в буфер.

Когда линейка становится доступна, запись применяется.

CPU должны видеть записи в буфере как в кэше.

Пример

```
 1 void foo(void)
 2 {
 3   a = 1;
 4   b = 1;
 5 }
 6
 7 void bar(void)
 8 {
 9   while (b == 0) continue;
10   assert(a == 1);
11 }
```

foo, b - CPU0, bar, a - CPU1

0: a -> store buffer, read invalidate a >

1: read b >

0: store b

0: read resp b==1 >

1: read resp b==1 <

1: while, assert

1: read invalidata a <

Решение

```
 1 void foo(void)
 2 {
 3   a = 1;
 4   smp_mb();
 5   b = 1;
 6 }
 7
 8 void bar(void)
 9 {
10   while (b == 0) continue;
11   assert(a == 1);
12 }
```

foo, b - CPU0, bar, a - CPU1

0: a -> store buffer, read invalidate a >

1: read b >

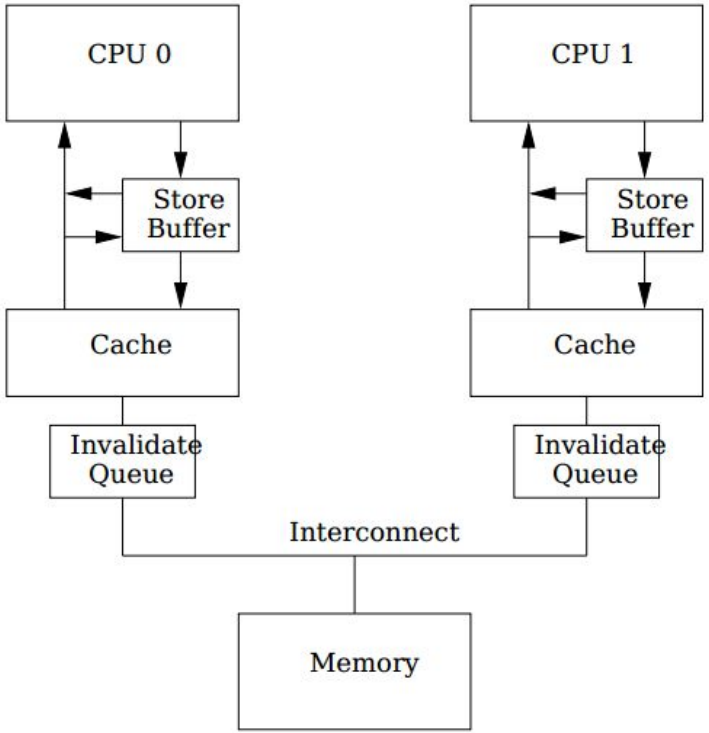
0: b -> store buffer

0: read resp b==0 >

1: read resp b==0 <

...

Буферы инвалидации



Записи из store buffer применяются по invalidation ack. Invalidation может быть долгим, если кэш загружен, например, invalidation сообщениями.

Поэтому, IACK высылается сразу, а invalidation запоминается

Проблема

```
 1 void foo(void)
 2 {
 3   a = 1;
 4   smp_mb();
 5   b = 1;
 6 }
 7
 8 void bar(void)
 9 {
10   while (b == 0) continue;
11   assert(a == 1);
12 }
```

b - M(cpu0), a - S

0: a -> store buf, inv a>

1: read b >

0: store b, read resp b >

1: inv a <, a -> inv q,

inv ack a >

1: read resp b <, read a

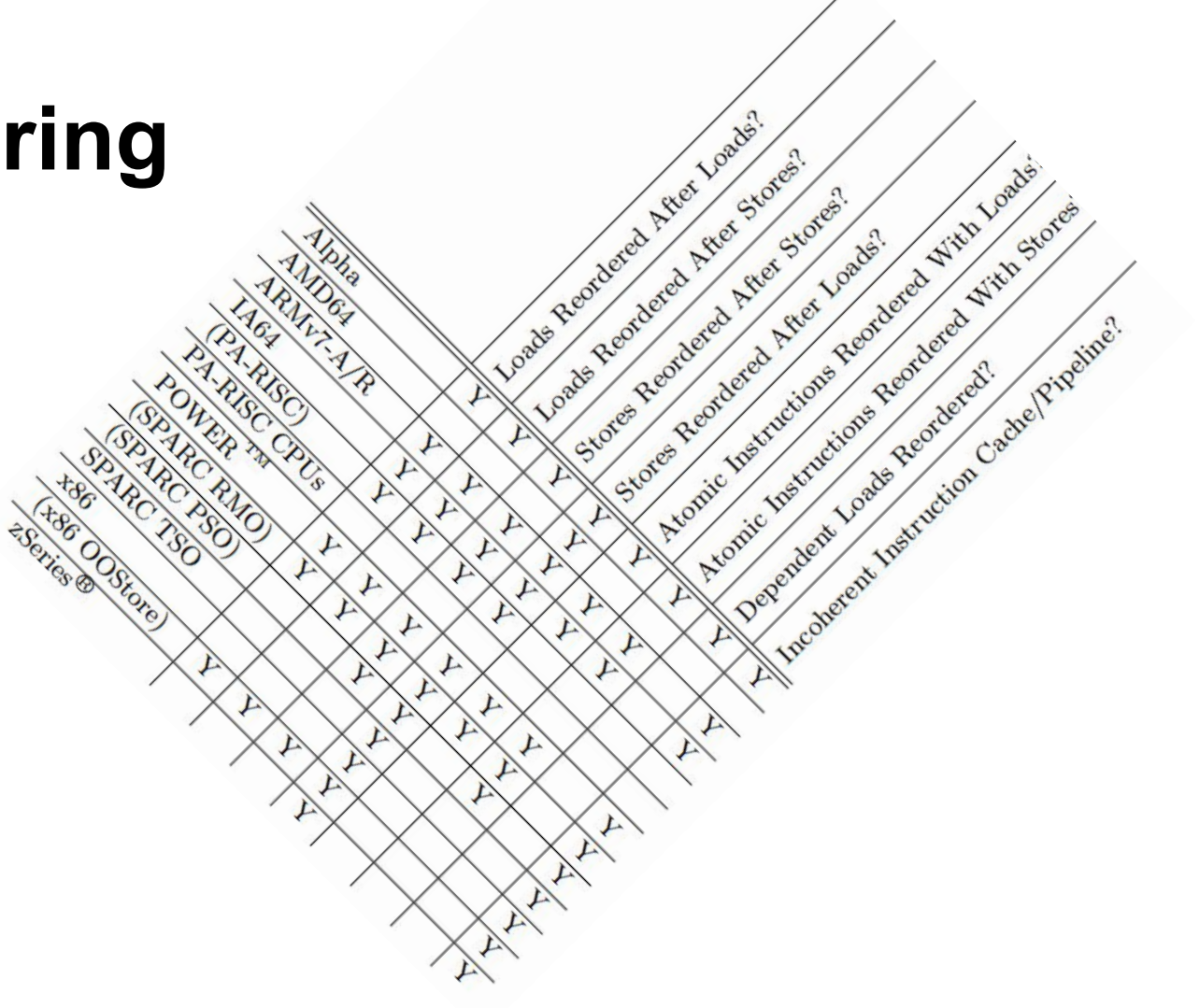
1: inv a apply

Решение

```
 1 void foo(void)
 2 {
 3   a = 1;
 4   smp_wmb();
 5   b = 1;
 6 }
 7
 8 void bar(void)
 9 {
10   while (b == 0) continue;
11   smp_rmb();
12   assert(a == 1);
13 }
```

Нужно применять весь invalidation buffer перед чтением *семантически* связанных областей

Ordering


Alpha
AMD64
ARMv7-A/R
IA64
(PA-RISC)
PA-RISC CPUs
POWER ™
(SPARC RMO)
(SPARC PSO)
SPARC TSO
x86
(x86 OOStore)
zSeries®
Loads Reordered After Loads?
Loads Reordered After Stores?
Stores Reordered After Stores?
Stores Reordered After Loads?
Atomic Instructions Reordered With Loads
Atomic Instructions Reordered With Stores
Dependent Loads Reordered?
Incoherent Instruction Cache/Pipeline?

Read-Copy-Update

https://www.kernel.org/doc/Documentation/RCU/whatisRCU.txt

Примитив синхронизации в Linux, для преобладающего чтения.

Пример

```
DEFINE_SPINLOCK(foo_mutex);

struct foo *gbl_foo;

int foo_get_a(void)       {
        int retval;
        rcu_read_lock();
        retval = rcu_dereference(gbl_foo)->a;
        rcu_read_unlock();
        return retval;
}
```

```
void foo_update_a(int new_a) {
        struct foo *new_fp;
        struct foo *old_fp;
        new_fp = kmalloc(sizeof(*new_fp),
GFP_KERNEL);
        spin_lock(&foo_mutex);
        old_fp = gbl_foo;
        *new_fp = *old_fp;
        new_fp->a = new_a;
        rcu_assign_pointer(gbl_foo, new_fp);
        spin_unlock(&foo_mutex);
        synchronize_rcu();
        kfree(old_fp);
}
```

Обзор API 1

- rcu_read_lock/unlock - критическая секция записи. Здесь запрещено блокирование. Могут быть вложенными или пересекаться
- synchronize_rcu - блокироваться, пока все предшествующие читатели в критической секции (не все читатели вообще)

Обзор API 2

- rcu_assign_pointer - присвоить новое значение указателю, защищенному RCU
- rcu_dereference - получить указатель из RCU-защищённого указателя

Концептуально

Обновление:

1. удаление (removal) ссылок на данные. Читатели видят либо старые, либо новые данные
2. ожидание, пока все читатели не выйдут из критических секций RCU чтения
3. утилизация (reclamation) старого ресура

RCU функции для указателей

```
#define rcu_assign_pointer(p, v) ({ \
                smp_wmb(); \
                (p) = (v); \
 })

#define rcu_dereference(p) ({ \
                typeof(p) __________p1 = p; \
                smp_read_barrier_depends(); \
                (__________p1); \
  })
```

Игрушечная реализация 1

```
static DEFINE_RWLOCK(rcu_gp_mutex);
void rcu_read_lock(void) {
        read_lock(&rcu_gp_mutex);
}
void rcu_read_unlock(void) {
        read_unlock(&rcu_gp_mutex);
}
void synchronize_rcu(void) {
        write_lock(&rcu_gp_mutex);
        write_unlock(&rcu_gp_mutex);
}
```

Игрушечная реализация 2

```
void rcu_read_lock(void) { }
void rcu_read_unlock(void) { }
void synchronize_rcu(void) {
        int cpu;
        for_each_possible_cpu(cpu)
                run_on(cpu);
}
```