Something Something Memory Ordering

喵喵

2022.1

Somewhere on the Internet...[1]

Wed, 09 Jun 2021 11:25:24 -0700

...

Look, memory ordering pretty much _is_ the rocket science of CS, but the C standards committee basically made it a ton harder by specifying "we have to make the rocket out of duct tape and bricks, and only use liquid hydrogen as a propellant".

[1] https://lwn.net/ml/linux-kernel/CAHk-%3DwgJZVjdZYO7iNbOhFz-iynrEBcxNcT8_u317J0-nzv59w%40mail.gmail.com/

Somewhere on the Internet...[1]

Wed, 09 Jun 2021 11:25:24 -0700

...

Look, memory ordering pretty much _is_ the rocket science of CS, but the C standards committee basically made it a ton harder by specifying "we have to make the rocket out of duct tape and bricks, and only use liquid hydrogen as a propellant".

Linus

[1] https://lwn.net/ml/linux-kernel/CAHk-%3DwgJZVjdZYO7iNb0hFz-iynrEBcxNcT8_u317J0-nzv59w%40mail.gmail.com/

What

你要写个锁：

```
bool locked;

// Aquire lock
while(atomic_swap(locked, true)) cpu_yield();

// Critical section
feed_cats()

// Release lock
locked = false;
```

What

O3 万事皆有可能!

What

O3 万事皆有可能！

其实不 O3 也有可能！

Why

Why

Decades-long 内卷 in μArch

Why

Decades-long 内卷 in μArch
General theme: 隐藏延迟

Why

Decades-long 内卷 in μArch
General theme: 隐藏延迟
但是延迟并未消失，会在 Least expect 的时候影响程序行为。

Why

Decades-long 内卷 in μArch
General theme: **隐藏延迟**
但是延迟并未消失，会在 Least expect **的时候影响程序行为。**

TL;DR: **因为缓存层级、多核心以及乱序执行互相背刺的结果**

Cache Hierarchy

延迟 vs. 面积 vs. 吞吐量

Cache Hierarchy

延迟 vs. 面积 vs. 吞吐量

多计算核心会导致问题：缓存不一致

Coherence Protocol

一般的实现：MESI 协议

Coherence Protocol

一般的实现：MESI 协议

- Modified: 本地存储有更改后的内容，兄弟节点不能拥有有效的这块儿数据
- Exclusive: 本地是唯一一份兄弟节点中有效数据
- Shared: 本地和兄弟节点共享这份数据
- Invalid: 本地没有这份数据

Coherence Protocol

一般的实现：MESI 协议

- Modified: 本地存储有更改后的内容，兄弟节点不能拥有有效的这块儿数据
- Exclusive: 本地是唯一一份兄弟节点中有效数据
- Shared: 本地和兄弟节点共享这份数据
- Invalid: 本地没有这份数据

需要某种方式传递 Coherence 信息

Coherence Protocol

一般的实现：MESI 协议

- Modified: 本地存储有更改后的内容，兄弟节点不能拥有有效的这块儿数据
- Exclusive: 本地是唯一一份兄弟节点中有效数据
- Shared: 本地和兄弟节点共享这份数据
- Invalid: 本地没有这份数据

需要某种方式传递 Coherence 信息

- Snooping: 一个 Coherence bus
- Directory: 一个 Coherence master

Interconnect

Problem: interconnect 有延迟

Interconnect

Problem: interconnect **有不稳定的**延迟

Interconnect

Problem: interconnect **有不稳定的**延迟

不同的 Coherence **信息可能在** Interconnect **网络内花不同的时间传递!**

Interconnect

Problem: interconnect **有不稳定的**延迟

不同的 Coherence **信息可能在** Interconnect **网络内花不同的时间传递!**
e.g. NUMA, distributed LLC, ...

Interconnect

Problem: interconnect **有不稳定的**延迟

不同的 Coherence **信息可能在** Interconnect **网络内花不同的时间传递!**
e.g. NUMA, distributed LLC, ...
为了维持更强的内存序，需要牺牲 Interconnect / NoC **的性能保序。**

E.g.

Core A

```
// Initially, *ptrA == *ptrB == false
*ptrA = true;
*ptrB = true;
```

Core B

```
int readA = *ptrA;
int readB = *ptrB;
assert(!(readA == false && readB == true));
```

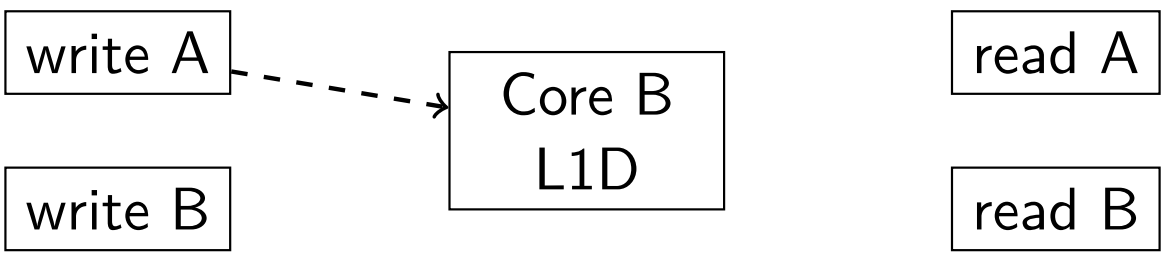
E.g.

Core A

```
// Initially, *ptrA == *ptrB == false
*ptrA = true;
*ptrB = true;
```

Core B

```
int readA = *ptrA;
int readB = *ptrB;
assert(!(readA == false && readB == true));
```



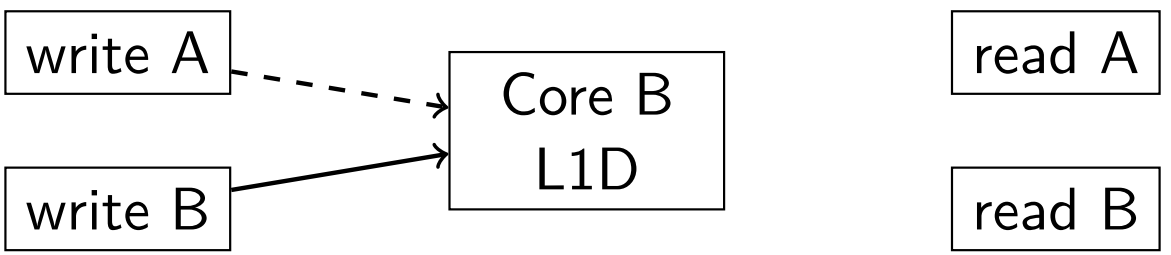
E.g.

Core A

```
// Initially, *ptrA == *ptrB == false
*ptrA = true;
*ptrB = true;
```

Core B

```
int readA = *ptrA;
int readB = *ptrB;
assert(!(readA == false && readB == true));
```



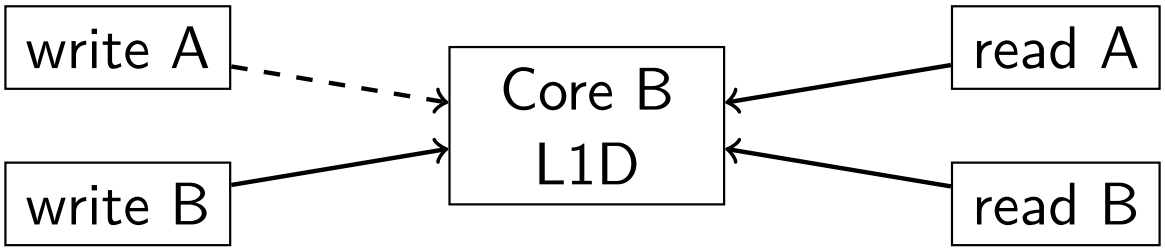
E.g.

Core A

```
// Initially, *ptrA == *ptrB == false
*ptrA = true;
*ptrB = true;
```

Core B

```
int readA = *ptrA;
int readB = *ptrB;
assert(!(readA == false && readB == true));
```



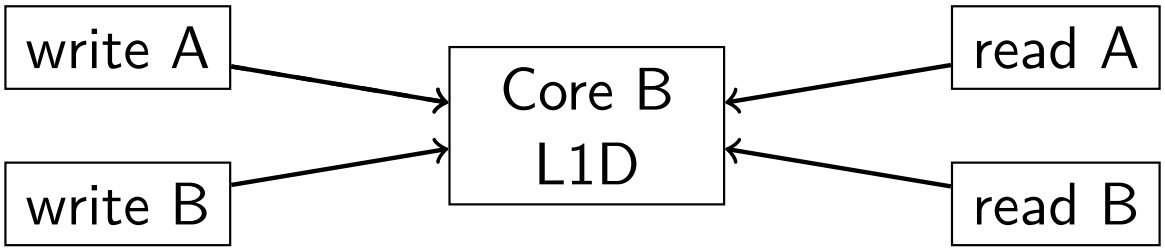
E.g.

Core A

```
// Initially, *ptrA == *ptrB == false
*ptrA = true;
*ptrB = true;
```

Core B

```
int readA = *ptrA;
int readB = *ptrB;
assert(!(readA == false && readB == true));
```



E.g. 2

Core A

```
*start = true;
```

Core B

```
while(!*start) // spin;
*end = true;
```

Core C

```
while(!*end) // spin;
assert(*start)
```

E.g. 3

Core A

```
*ptrA = true;
```

Core B

```
*ptrB = true;
```

Core C

```
while(!*ptrA) ;
while(!*ptrB) ;
*cnt += 1;
```

Core D

```
while(!*ptrB) ;
while(!*ptrA) ;
*cnt += 1;
```

Also be careful for...

- Store buffers
- Victim FIFO

Out-of-Order Pipeline

Welcome to the mess^{TM}.

Out-of-Order Pipeline

Welcome to the mess^{TM}. 现在 L1D 看到的访存和真正的访存顺序都可以不一样了!

Out-of-Order Pipeline

Welcome to the mess^{TM}. 现在 L1D 看到的访存和真正的访存顺序都可以不一样了!

一个常见优化：非阻塞缓存——允许后发生的，没有 Miss 的访存请求先返回。

What's the cost...

为了追逐~~力量~~进行微架构优化，我们使得：

- 访存指令发射的顺序可能和真实顺序不同
- L1D 处理访存的顺序可能和指令发射的顺序不同
- 其他节点接收到 Coherence 消息的顺序可能和发出的顺序不同，或者不一致

To regain sanity...

定义 Memory order: **一些访存操作的额外属性，约束编译器和微架构实现的优化。**

To regain sanity...

定义 Memory order: **一些访存操作的额外属性，约束编译器和微架构实现的优化。**

- Sequence Consistent：**要求所有带有这一属性的访存在全局有一个一致的观测顺序。**
- Acquire-Release **语义**：**对应常见的锁的实现**
  - Acquire: **（通常作为读取）其后的访存不允许重排到其之前。**
  - Release: **（通常作为写入）其前的访存不允许重排到其之后。**
- Relaxed: **代表没有任何额外要求**

To regain sanity...

定义 Memory order：一些访存操作的额外属性，约束编译器和微架构实现的优化。

- Sequence Consistent：要求所有带有这一属性的访存在全局有一个一致的观测顺序。
- Acquire-Release 语义：对应常见的锁的实现
  - Acquire: （通常作为读取）其后的访存不允许重排到其之前。
  - Release: （通常作为写入）其前的访存不允许重排到其之后。
- Relaxed: 代表没有任何额外要求

通常为了保证软件不需要滥用这些访存约束，ISA 通常会给出额外的一些约束，放弃不重要的优化，使得常见的访存模式可以给出符合直觉的结果。

To regain sanity...

定义 Memory order：一些访存操作的额外属性，约束编译器和微架构实现的优化。

- Sequence Consistent：要求所有带有这一属性的访存在全局有一个一致的观测顺序。
- Acquire-Release 语义：对应常见的锁的实现
  - Acquire：（通常作为读取）其后的访存不允许重排到其之前。
  - Release：（通常作为写入）其前的访存不允许重排到其之后。
- Relaxed：代表没有任何额外要求

通常为了保证软件不需要滥用这些访存约束，ISA 通常会给出额外的一些约束，放弃不重要的优化，使得常见的访存模式可以给出符合直觉的结果。
e.g.: 要求同一个核心的 RAW 访问符合直觉。

Case: RVWMO & Tilelink & BOOM

Case: RVWMO & Tilelink & BOOM

~~我很想讲 BOOM 但我不太会，GG~~

Case: RVWMO & Tilelink & BOOM (疑似)

RVWMO TL;DR:

- 默认核心内部 Strong order, 核心之间 Weak order
  - 额外的，不允许 L-L 重排
- Acquire-Release 语义 + Sequential consisten 要求
- LL/SC + AMO

L-L 重排

```
ld a0, 0(s0)
ld a1, 0(s0)
// a0 NOT ALLOWED to get newer versions than a1
```

Case: RVWMO & Tilelink & BOOM (疑似)

Tilelink TL;DR:

- 节点拓扑构成有向无环，单一路径的图，方便维护 Coherence 状态。节点需要维护孩子的 Coherence 状态。
- 经典的 MESI + 扩展（见下图）
- 典型实现中，AMO 操作在 Coherence master (LLC) 处发生

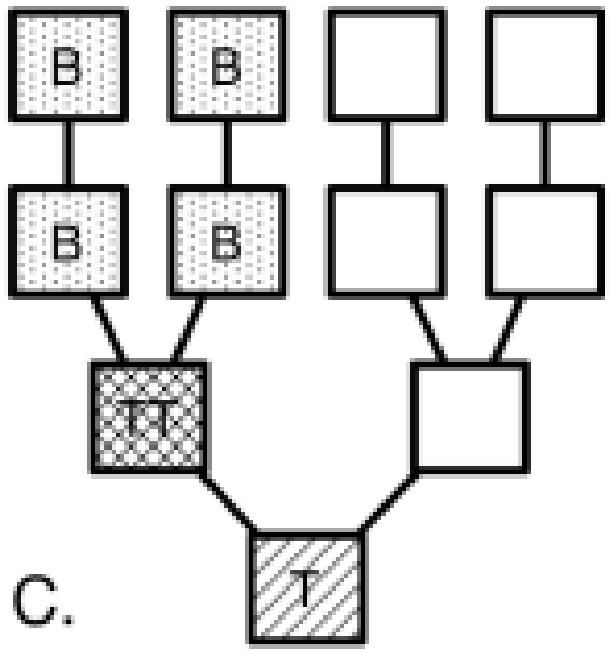


图: TileLink 节点 Coherence 状态

Case: RVWMO & Tilelink & BOOM (疑似)

Tilelink 这样的设计允许的优化 e.g.:

- 对应 RVWMO，对核心和缓存的实现比较方便
- 修改的版本存储在最接近访问的地方，减少 Writeback 所用的带宽
- 用于同步的内存访问发生在 LLC，减少 Coherence message 所用的带宽

Case: RVWMO & Tilelink & BOOM (疑似)

Tilelink 这样的设计允许的优化 e.g.:

- 对应 RVWMO，对核心和缓存的实现比较方便
- 修改的版本存储在最接近访问的地方，减少 Writeback 所用的带宽
- 用于同步的内存访问发生在 LLC，减少 Coherence message 所用的带宽

禁止的优化 e.g.:

- 不允许多条路径，因此无法实现 Distributed L2 / LLC

Case: RVWMO & Tilelink & BOOM (疑似)

BOOM (疑似) TL;DR:

- 是一个乱序处理器！积极发射所有的访存，遇到违反核心内部顺序或者 L-L 重排时发送微架构异常刷新流水线。
- 非阻塞 Cache

Case: RVWMO & Tilelink & BOOM (疑似)

BOOM (疑似) TL;DR:

- 是一个乱序处理器！积极发射所有的访存，遇到违反核心内部顺序或者 L-L 重排时发送微架构异常刷新流水线。
- 非阻塞 Cache

如果允许更弱的内存序，可以实现更多优化：

- 允许核心内访存重排：完全不检查任何访存乱序

Other Considerations

- Speculative execution
- Prefetch
- I$ Coherence?
- Non-explicit memory access (page table, A/D bits)

RISC-V ISA 的其他相关细节

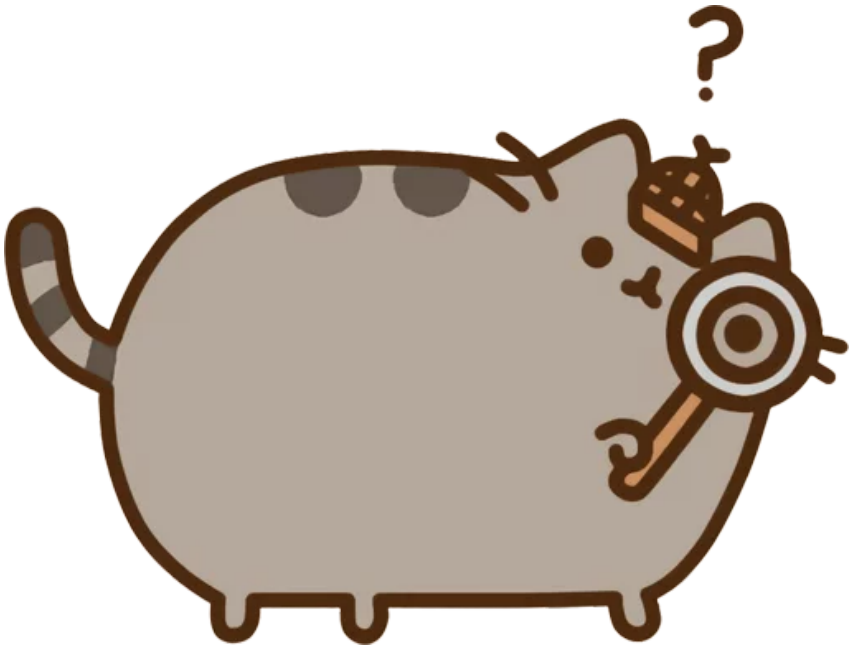
允许的优化：

- 显式 FENCE.I / SFENCE.VMA，不需要 Coherence I$ / TLB
- PMA 只有部分内存需要支持 AMO (对应 TileLink 的特定 Message)

导致的实现要求

- 页表访问遵循 RVWMO，直接导致所有的实现基本都将 PTW 连接到 D$ 上

That's All!



Question time!