Precise Exceptions & Speculation

Precise Exceptions

Exceptions vs. Interrupts

造成的原因

- 异常：当前线程内部（一般是指令造成）
- 中断：当前线程外部（IO 设备就绪等）

何时处理

- 异常：检测到就要处理
- 中断：方便时处理，除了掉电等非常紧急的情况

处理的上下文

- 异常：进程
- 中断：系统

Precise Exceptions

要求：架构状态必须在处理中断/异常时保持一致

- 中断之前的指令必须完全 retire
- 中断之后的指令不应当 retire

> retire：执行结束，并且更新了 AS

精确中断源于冯诺依曼架构本身语义的要求：指令按序执行

可以轻易地从异常中恢复，也可以支持可重启的进程

朴素的方法就是让所有指令的延时都相同

Reorder Buffer (ROB)

思路：指令在完成时是乱序的，但是在更新 AS 之前将其组织为有序

- 当指令解码时，在 ROB 中保存一个条目
- 当指令完成时在 ROB 中写结果

- 如果指令是 ROB 中最老的指令，且完成前没有发生异常/中断，则将其结果写入寄存器/内存

在 Tomasulo 中，使用 ROB 的编号来代替 RS，ROB 也可以作为重命名的寄存器提供操作数，当指令 commit 时再写入寄存器

使用 ROB 可以在预测错误或中断时轻易地撤销预测执行的指令

Tomasulo with ROB 可以实现按序 issue，乱序执行并完成，然后再按序 commit

Handling Speculation and Exception

ROB 可以支持 speculation，等到预测的 branch 到达 commit 阶段，如果 BP 预测错误，则冲刷 ROB，RS 和寄存器状态，然后从正确的 target 开始（等到 branch commit，则其之前的指令都已 retire 完成）

如果发生异常，在 ROB 中增加一个 indicator，直到 commit 再处理异常

Tradeoff for ROB

优点

- 概念上来说比较简单
- 消除了错误的依赖，支持精确中断

缺点

- ROB 需要大量读写，且会增长 critical path
- 过多的复制操作：寄存器堆-RS-ROB-寄存器堆
- RS 混合了数据和控制

Other solutions

History buffer：指令完成时更新寄存器堆，异常产生时则撤销之前的更新

- 解码时给指令保留一个 HB 的条目
- 指令完成时将目的寄存器的旧值保存在 HB
- 如果指令是当前最旧的指令并且没有异常/中断发生，丢弃其 HB 条目，否则将其余 HB 的值从尾到头写回原本的寄存器

Future register file：维护两套寄存器，speculative 和 architectural，future file 用于获取最新的寄存器值，而 architectural file 用于异常发生时恢复状态

Checkpointing：保存 branch 解码时的状态，如果预测错误则根据此恢复

比较上述各种处理方式

- ROB

    - 源值可能在 ROB 或是寄存器堆，数据访问复杂
    - ROB 读写在 critical path
- HB
    - history buffer 读不在 critical path（异常发生时才读）
    - 需要读取寄存器旧值
- Future file
    - future file 可以最快地访问最新的值
    - 需要维护两套寄存器堆

Memory Access in OoO

访存指令同样可以享受乱序执行的优点，尤其是 load 指令。

load 指令应当尽早执行，因为在数据流图中 load 处于顶端（地址确定后不依赖于其他指令，但是多数指令依赖 load 的结果）

Memory Dependency

与寄存器指令相同，内存指令同样有三种数据依赖：RAW, WAW, WAR，然而这些依赖与寄存器的依赖有根本的不同

- 寄存器依赖是静态可知的，但是内存依赖是动态决定的，看指令难以判断
- 寄存器状态空间小，内存状态空间大
- 寄存器状态对其他线程/处理器不可见，内存状态对其他线程/处理器可见

内存依赖的问题是内存地址直到 load/store 执行时才可见

- 难以对内存地址重命名
- 依赖关系只有在部分执行后才能确定
- 当一条 load/store 确定地址时，可能有更早的指令还未确定地址

Basic Concepts

memory aliasing：两个访存涉及到同一内存位置

memory disambiguation：确定两次访存是不是会 alias，需要高效地计算地址

load bypass：如果 load 与更早的 store 都没有 alias，则可以提前执行

load forwarding：如果 load 与其之前的某个 store 地址匹配，则可以直接使用 store 的值

有多种方式可以 memory disambiguation

- 假设所有 load 都与之前的 store 有依赖

    - 不需要恢复
    - 过于保守，会带来不必要的延迟
- 假设所有 load 都与之前的 store 独立
    - 简单，不需要延迟
    - 当有依赖时需要恢复以及再运行
- 预测 load 的依赖
    - 更精确，且依赖关系不会随时间改变
    - 仍然需要预测错误时的恢复和再执行

In-Order load/store

按照相对顺序执行 load/store，但是可以与其他非内存指令乱序执行

即假设所有内存指令间都有依赖

但是不同指令完成的阶段不同

- load 在 execute 阶段就完成
- store 要等到 retire 阶段

LSQ (Load/Store Queue) 是一个环形的队列，指令按照程序顺序存储（issue 时分配，retire 时去配），对于每个指令存储

- 类型：load 或是 store
- 地址：内存地址，地址的生成按照乱序，生成时拷贝到 LSQ
- 值：待存的值

LSQ 可以看作内存的 RS

只有在 LSQ 头部而且就绪的指令才能执行

- 如果是 load，准备好就可执行
- 如果是 store，只有其同时是 ROB 的头部且就绪时才能执行

因为 store 执行了就会改变 AS，所以需要等到其之前的指令均已 retire

按序的 load/store 易于实现但是影响性能

其流水线可以总结为

- store
    - Issue：在 LSQ 尾部分配条目
    - Execute：计算地址，将待存数据存入 LSQ 条目
    - Retire：将数据写入 D cache，然后释放 LSQ
- load
    - Issue：在 LSQ 尾分配条目

    - Address gen：计算地址，写入 LSQ
    - Execute：读 D cache
    - Retire：释放 LSQ 头

Load Forwarding + Bypassing

如果没有 alias，load 可以提前执行，且 load 可以从更早的地址相同的 store 直接获得数据

- 需要检查之前的 store 的地址
- 之前的 store 地址必须可知

实现时需要分离的 LQ (Load Queue) 和 SQ (Store Queue)

且需要知道队列中指令的相对顺序：添加新的字段 age，使用一个简单的随 issue 增加的 counter 即可

对于 LQ 中最老的 load，需要扫描检查 SQ 中的地址，如果有一条更老的 store 地址未知或与其地址相同，则 load 不能执行，如果地址相同的 store 数据可用，则 load 直接从这里获得数据

需要 CAM (associative memory) 结构来根据地址查询

load 同时查询 D cache

store 只要位于 ROB 头即可执行

其流水线可总结为

- store
    - Issue：在 SQ 尾部分配条目，记录 age
    - Execute：计算地址，将地址和数据写入 SQ
    - Retire：将数据写入 D cache，然后释放条目
- load
    - Issue：在 LQ 尾部分配条目，并记录 age
    - Address gen：生成地址，写入 LQ
    - Execute：向 D cache 发出请求，并且检查 SQ 有没有 alias 出现
    - Retire：释放 LQ 条目

Execute Load when Ready

之前的策略缺点在于 load 必须等待所有之前的 store 计算出地址

可以采用更激进的策略：即使有地址未知的 store，仍执行 load，这样 load 永不 stall

这样的依据是 alias 并不常见，但是推测错误的话需要撤销错误的 load

当旧的 store 在新的 load 之后执行时

- 检查所有更新的 load
- 如果地址匹配，则产生了 violation
- 需要 LQ 也支持 CAM

其流水线可描述为

- store
  - Issue：在 SQ 尾部分配条目，记录 age
  - Execute：计算地址，将地址和数据写入 SQ
  - Retire：将数据写入 D cache，释放 SQ 条目，检查 LQ 是否有 alias，如果有则恢复
- load
  - Issue：在 LQ 尾部分配条目，并记录 age
  - Address gen：生成地址，写入 LQ
  - Execute：向 D cache 发出请求
  - Retire：释放 LQ 条目

需要注意的是，会出错的不仅是 load，如果 load 出错，错误的值会随着后续指令传播

简单的方法是将错误的 load 之后的所有指令都冲刷掉重新执行，复杂的方法是只冲刷有依赖的指令

Performance

简单的预测器（根据过去的历史）来预测 store-load 的依赖可以达到最好的效果

Summary for Superscalar

Questions to Ponder

为什么乱序执行是有用的：指令的延时不同，乱序执行可以在执行高延时的指令时

乱序执行能容忍多大的延时

有那些设计上的 tradeoff

- 更高的主频还是更好的 OoO 引擎?
- 数据的值应该存在 ROB 和 RS 还是一个中心化的物理寄存器堆
- RS 应该是中心化的还是分布在各个 function unit
- RS 和 ROB 有多少条目

Beyond ILP

乱序执行的超标量处理器可以使得顺序程序得以 ILP

但是 IPC 一般受限于

- 有限的带宽，包括：内存，cache，fetch 和 commit，重命名等
- 有限的硬件资源：寄存器数量，ROB/RS/LSQ 的条目数，function unit
- 真正的数据依赖，即程序中不可并行的部分
- BP 准确度
- memory disambiguation

随着系统越来越复杂，会逐渐产生边际效应，性能的提升越来越有限

User Visible/Invisible

一开始的并行是用户不可见的，由微架构实现顺序程序的并行

用户只需要编写程序，然后买更好的 chip 就有性能提升

但是现在用户也需要编写可并行的程序以最大程度利用硬件，这样可以解放硬件，从而获得更简单高效的硬件

- TLP：多处理器，硬件实现的多线程
- DLP：向量机，SIMD，GPU
- RLP：数据中心