chap 3.3 - 5 笔记

高文才 20023012

November 23, 2020

1 利用高级分支技术降低分支代价

分支可能会产生控制依赖，从而降低流水线性能。循环展开是一种解决方式，此外还有下面的解决办法。

1.1 相关分支预测器 (Correlating Branch Predictors)

2 位预测器只是利用单一分支指令的历史来预测，显然利用他与其他分支指令之间的相关性来预测会提高预测精确度，考虑如图 1的例子。显然，第三个 if 语句与前两个 if 语句之间是有相关性的，如果前两个语句成立则会使：$aa = bb = 0$，那么

```
if (aa==2)
        aa=0;
if (bb==2)
        bb=0;
if (aa!=bb) {
```

Figure 1: 分支代码之间存在相关性的示例

第 3 个 if 语句一定不成立。

利用其他分支结果的预测器叫作相关预测器 (correlating predictors) 或者叫作两级预测器 (two-level predictors)，记作 (m, n), 表示从最近的 m 个分支的 2^m 个分支预测器的行为进行预测，每个单一分支对应预测器由 n 位组成。这样一个 (m, n) 预测器的总位数应该是 $2^m * n * entries$，一个 2 位预测器就是一个不带有全局历史的 $(0, 2)$ 预测器。图 2是一个著名的关联预测器的例子叫 gshare 预测器，它的索引由分支地址和最近的分支结果进行异或而成，代表着对分支地址和分支历史的散列。

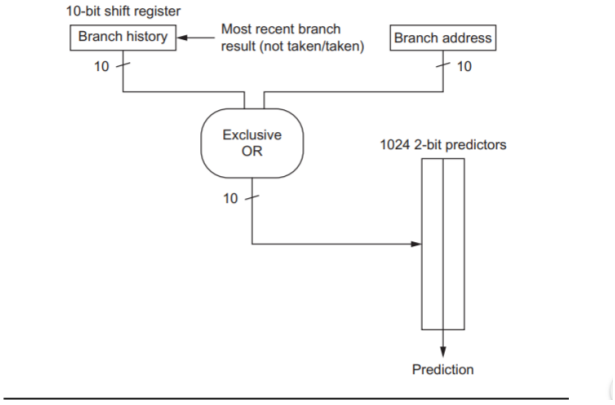

Figure 2: 一个 1024 个入口，每一个都是一个标准的 2 位预测器的例子

1.2 锦标赛预测器 (Tournament Predictors)

锦标赛预测器的优点是利用多个预测器，例如一个全局预测器和一个局部预测器，利用一个选择子进行选择，全局预测器利用最近的分支历史来索引，局部预测器利用地址来索引，图 3是它的例子。

1.3 Tage 预测器 (Tagged Hybrid Predictors)

基于 PPM(Prediction by Partial Matching) 思想的 Tag 预测器。它的一个例子如图 4。它总是会取 tag 和 hash 命中的最长的全局分支历史的预测器的预测结果，即越靠右边的结果，而最左边就是默认的由 pc 直接索引的饱和计数器。

2 动态调度 (dynamic scheduling)

如果数据依赖不能解决，前面的方法都只有暂停流水线，这小节将利用动态调度，一种利用硬件记录指令执行的方法在保持数据流正确的情况下减少暂停。它有以下几点好处：

- 只需编译一次，可以高效运行
- 可以处理在编译时不确定的依赖
- 它允许存在不确定的延迟，例如 cache 缺失

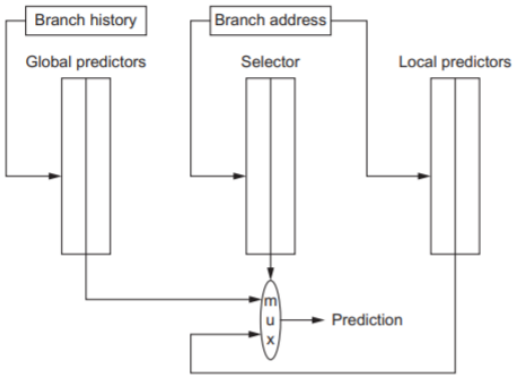

Figure 3: 一个锦标赛预测器的例子。

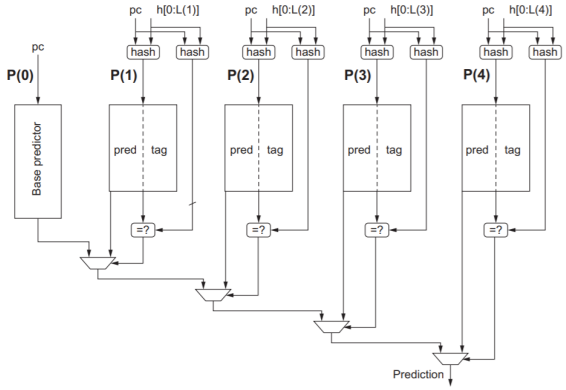

Figure 4: 一个 tage 预测器的例子。

2.1 基本思想

考虑以下代码：

- fdiv.d f0,f2,f4
- fadd.d f10,f0,f8
- fsub.d f12,f8,f14

fadd.d 与 fdiv.d 之间存在数据依赖，那么后面的 fsub.d 也必须暂停，尽管实际上前两条指令对它根本没有什么影响。

为了解决上述问题，可以先将 issue 阶段分为两个部分：结构冲突检查和数据冲突检查。我们仍然保持顺序发射，但我们期待可以乱序执行。

乱序发射可能会带来 WAR 和 WAW 冲突，例如考虑以下代码：

- fdiv.d f0,f2,f4
- fmul.d f6,f0,f8
- fsub.d f0,f10,f14

如果 fsub.d 先于 fmul.d 执行，则会产生 WAR 冲突。

另外，动态调度处理器可能会产生不精确异常。

区别于记分牌算法，下面介绍的托马苏洛算法将会依靠动态寄存器重命名来高效的消除反相关和输出相关。

2.2 托马苏洛算法 (Tomasulo's Approach)

尽管这种方法有很多变体，两个最核心的思想分别是：动态决定一条指令是否可以执行 (消除 RAW) 与寄存器重命名来避免不必要的冲突 (消除 WAR 和 WAW)。

下面的左右两段代码显示了潜在的 WAR 和 WAW 冲突是如何通过寄存器重命名消除的 (注意观察关于 f8 的 WAR 与关于 f6 的 WAW):

- fdiv.d f0,f2,f4 fdiv.d f0,f2,f4
- fadd.d f6,f0,f8 fadd.d S,f0,f8

- fsd f6,0(x1) fsd S,0(x1)
- fsub.d f8,f10,f14 fsub.d T,f10,f14
- fmul.d f6,f10,f8 fmul.d f6,f10,T

在托马苏洛算法中，保留站与功能部件相连且提供了寄存器重命名功能。它的精华在于保留站将会缓存或者读取操作数，越快越好，从而避免需要从寄存器中读取数据造成冲突。

保留站的使用带来了以下两个重要特性：1. 分布式的冲突检测和控制执行；2. 结果是直接从缓存数据的保留站中送到功能部件，而不是经过寄存器，这样的实现允许所有的等待部件同时读取同一份数据 (通过 CDB)。图 5展示了该算法的基本结构。

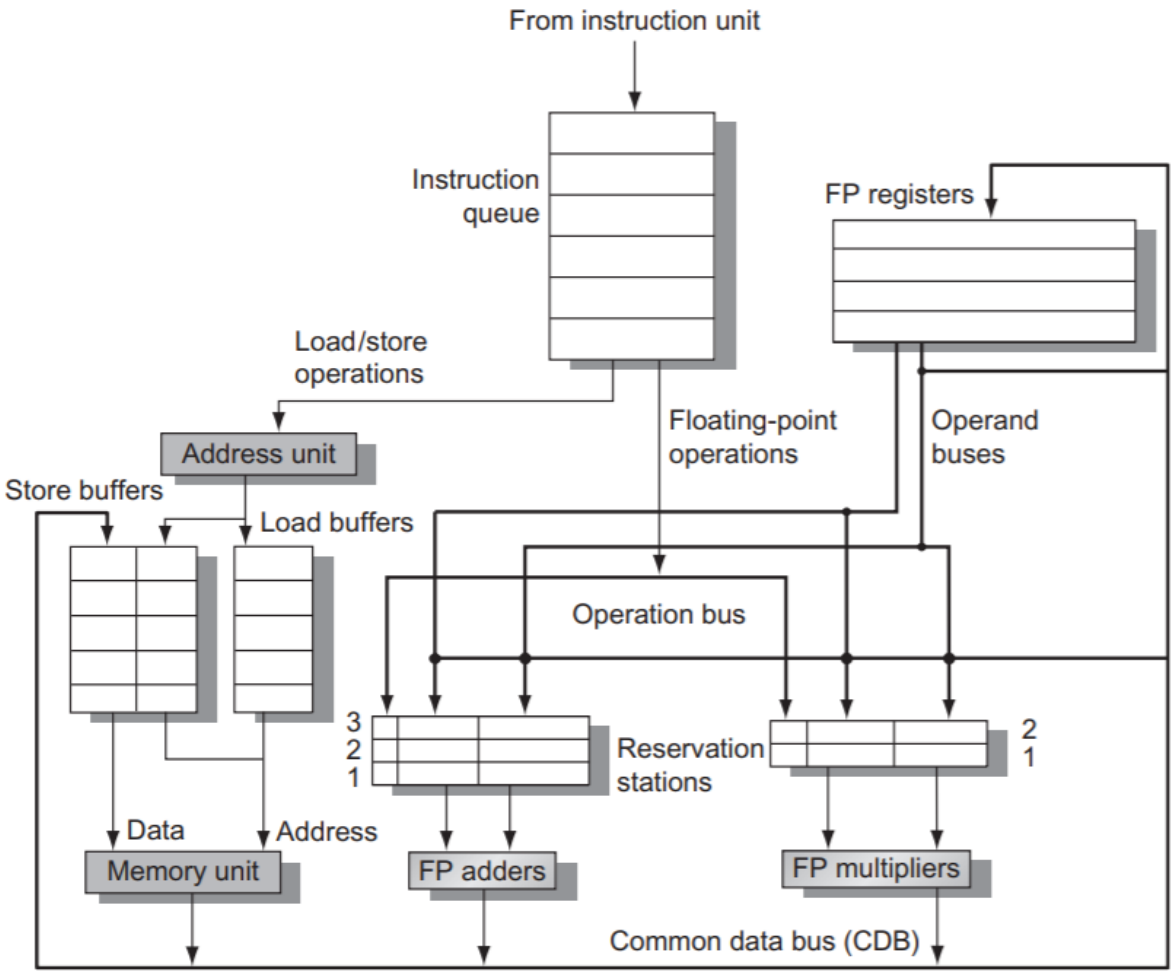

Figure 5: 基本的托马苏洛算法结构图

算法的实现主要包括以下三个步骤：

- Issue 从队列取指令，当前仅当对应的保留站有空闲 (避免结构冲突)，且所有的操作数都已就绪才派遣该指令，否则就一直检测直到产生数据的功能部件都已确定 (在这个阶段进行寄存器重命名，消除 WAR 和 WAW)。
- Execute 如果当前运算还有操作数没就位，那么必须等待。通过延迟指令的执行可以消除 RAW。值得注意的是，由于是乱序执行，在允许分支开始执行的之后的指令之前，处理器必须知道分支预测是正确的。
- Write result 结果产生后，广播到 CDB 上，各个部件按需使用。

保留站的基本数据结构如下：

- Op 对应的操作
- Qj,Qk 产生两个源操作数的功能部件
- Vj,Vk 两个源操作数的值，V 域和对应的 Q 域只会有一个有效
- A 在有效地址计算产生后记录在此 (针对 load 与 store 指令)，
- busy 有效位

寄存器堆的每个寄存器都有一个 Qj 域，用来保存哪个功能部件产生的数据会保存在这个寄存器。

3 例子

3.1 例子

例子略，在教材上或者 ppt 上可以查看。

3.2 算法细节

图 6 展示了算法的每个步骤应该进行的操作以及每个数据项应该如何更新。

Instruction state	Wait until	Action or bookkeeping
Issue FP operation	Station r empty	if (RegisterStat[rs].Qi≠0) {RS[r].Qj ← RegisterStat[rs].Qi} else {RS[r].Vj ← Regs[rs]; RS[r].Qj ← 0}; if (RegisterStat[rt].Qi≠0) {RS[r].Qk ← RegisterStat[rt].Qi else {RS[r].Vk ← Regs[rt]; RS[r].Qk ← 0}; RS[r].Busy ← yes; RegisterStat[rd].Q ← r;
Load or store	Buffer r empty	if (RegisterStat[rs].Qi≠0) {RS[r].Qj ← RegisterStat[rs].Qi} else {RS[r].Vj ← Regs[rs]; RS[r].Qj ← 0}; RS[r].A ← imm; RS[r].Busy ← yes;
Load only		RegisterStat[rt].Qi ← r;
Store only		if (RegisterStat[rt].Qi≠0) {RS[r].Qk ← RegisterStat[rs].Qi} else {RS[r].Vk ← Regs[rt]; RS[r].Qk ← 0};
Execute FP operation	(RS[r].Qj = 0) and (RS[r].Qk = 0)	Compute result: operands are in Vj and Vk
Load/storestep 1	RS[r].Qj = 0 & r is head of load-store queue	RS[r].A ← RS[r].Vj + RS[r].A;
Load step 2	Load step 1 complete	Read from Mem[RS[r].A]
Write result FP operation or load	Execution complete at r & CDB available	∀x(if (RegisterStat[x].Qi=r) {Regs[x] ← result; RegisterStat[x].Qi ← 0}); ∀x(if (RS[x].Qj=r) {RS[x].Vj ← result;RS[x].Qj ← 0}); ∀x(if (RS[x].Qk=r) {RS[x].Vk ← result;RS[x].Qk ← 0}); RS[r].Busy ← no;
Store	Execution complete at r & RS[r].Qk = 0	Mem[RS[r].A] ← RS[r].Vk; RS[r].Busy ← no;

Figure 6: 托马苏洛算法的实现细节，每个阶段的执行步骤