奔腾家族处理器的分支预测

描述了 Pentium 的分支预测机制，和以后版本的更有效的分支预测功能。

作者：Agner Fog

翻译：coly li

什么是分支预测

想象一个简单的微处理器，他的所有的指令都分为两个阶段被处理：解码，和执行。这个处理器可以通过在执行一条指令的时候同时界面下一条指令的这种方式来节省时间。这种流水线式（assembly line－principle）的原理被称为管线(pipeline 也有人翻译为流水线)。管线可以有很多阶段，这样很多连续的指令可以同时运行在流水线中，每一个指令处在管线的一个阶段中。

当我们遇到一个分支指令的时候就会遇到一个问题。分支指令是一个 if-then-else 结构的实现。如果这个结构是真，那么指令执行就跳转到一个别的地方，如果结构为假，则继续执行下一条指令。这就将管线中存在的指令们打断了，因为处理器不能预先知道下一条应该执行的指令是什么，除非执行完分支指令。管线越长，处理器就要等待更长的时间，直到他知道下一条进入管线的指令是什么。鉴于现在的微处理器趋向于采用越来越长的管线，所以解

决由于分支指令导致管线失败问题的需求也就增长了。

解决方案就是分支预测。微处理器会基于一个记录该分支先前的跳转结果的记录尝试去预测分支指令这次是否会导致指令跳转。如果前 4 次这个分支指令都导致了指令跳转，那么这一次发生跳转的几率就非常大。微处理器在知道真正的指令之前，就基于这种预测方式来确定下一条应该加载到管线中的指令。这种方式成为“投机执行”。如果这次预测是错误的，那么 CPU 就会清除管线并忽略所有基于这次错误的预测的计算结果。但是如果这次预测正确了，那么将会节省很多时间。

侦探工作

Intel 的手册从来没有精确的描述过分支预测是如何工作的。但是，由于分支预测错误将会增加大量的执行时间，我发现了解分支预测工作机制对于优化自己的程序非常重要。

我开始同我在 internet 上遇到的一些聪明人一起做了大量的试验，这些人中最重要的是科罗拉多大学的 Karki J. Bahadur，和挪威的 Terje Mathisen，他们将系统软件反向工程后发现了如何访问在 Pentium 芯片上的性能监视器计数器。我的第一个发现就是如果 Pentium 如果发现一个分支指令在最近两次执行中的任意一次都发生了跳转的话，它就会预测这一次也会发生跳转。

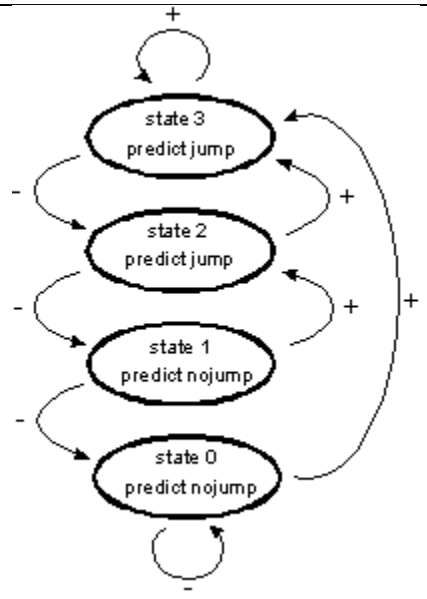
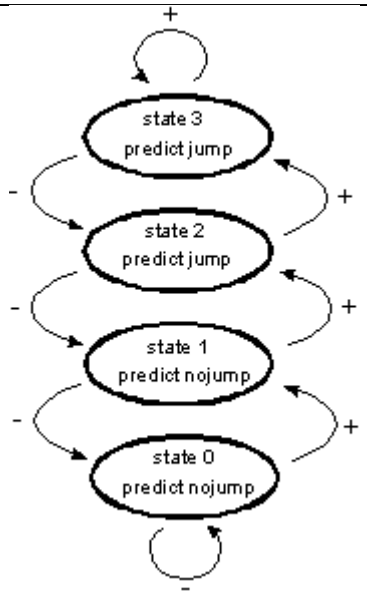
这符合我的所用试验。但是 Karki 指出，对于一个 branch 指令，如果其实际执行为每三次

跳转一次，那么分支预测器在对此 branch 进行 6 此预测当中，实际上只能够预测器中的一次跳转。也就是说，如果分支预测器的预测精度为 100%的话，它在对此 branch 进行 6 此预测当中，应该能够预测出两次跳转。（上面 Karki 指出的这段话，我翻译不过来，是从网上提问后采取 lus_wdh2 朋友的结论）

因此，根据我的模型，他永远不会预测正确。随着一系列新的试验，Karki 和我独立的推导出如图 fig 1a 中的状态图。当我们就这个机制达成一致的时候，我们在一个细节上的解释没有达成一致，这就是为什么这个图是不对称的？在这期间，另一个家伙在 Microprocessor Report 上发现了一篇旧论文，说明了这个预测机制是如 fig 1b 中所述的对称机制。我的观点是设计者实际上是比较倾向于 fig 1b 所述的分支预测机制的，而这个不对称的状态是一个 Bug。但是 Karki 和 Terje 仍然坚持这种不对称的机制后面一定有一个特定的意图。

虽然我给出示例说明对称的机制在几乎所有的情况下都优于不对称的机制，但是这仍然不能使他们信服。

图 1——分支预测机制的状态图

	
a，Pentium 中的不对称设计 当发生分支跳转时，状态跟随+箭头，当没有发生分支跳转时，状态跟随-箭头。 当处于状态 2 或状态 3 时，分支指令将被预测为跳转，如果处于状态 0 或状态 1 时，则被预测为不跳转。	b，对称设计 这是分支预测应当的工作模式。当跳转发生时状态增加，当跳转没有发生时，状态减少。

现在我发现了一个强大的工具去深度挖掘这个机制。Pentium 有一个测试寄存器集，通过他们就可以直接读写保存有所有的分支的历史信息的区域，这个区域叫做 Branch Target Buffer(BTB)。我在另一个黑客 Christian Ludloff 的个人主页上看到了这个信息。他的网页被砍掉了（谣传是迫于 Intel 的压力），但是我有幸及时的下载了他的网页。

通过直接访问 BTB，我可以准确的看到发生了什么：当 BTB 中没有某一个分支的记录项时，它就被预测为不跳转。当他第一次发生跳转时，它就会从 BTB 中获取一个记录项，并立刻转入状态 3。这是因为设计者将状态 0 视为” vacant BTB entry”。

这是有意义的，因为状态 0 永远都被预测为跳转。但是，由于它没有办法区分 State 0 和” vacant BTB entry”，所以当下一次的分支发生跳转后，它就会转入状态 3，而不是状态 1。

这就是为什么这里看起来比较奇怪。显然在设计实验室中的某些人做了大量的试验和研究来发现一个较好的分支预测策略，然后另外一些人将状态 0 同” vacant BTB entry”混淆了以至于没有实现这个策略。所以当这个策略遇到一个随机跳转的分支时，发生预测错误的几率将会比对称设计的大 3 倍。

Karki 和 Terje 仍然深信这不是一个设计的大错误。但当我发现一个紧循环（ tight loops ）会发生不一样的行为时，令人信服的证据有了。在一个小循环中微处理器没有足够的时间去在下一次遇到相同分支指令之前升级 BTB 记录项。为了避免延迟，CPU 绕过了 BTB 而是从管线中直接读取分支预测状态。在这种情况下它可以从状态 0 转到状态 1，就像在 fig 1b 中那样。

更怪僻的

我们很快发现在 Pentium 的分支预测中还有更奇怪的东西。我们不知道当一个分支指令紧跟着一个分支指令会发生什么。这一次 Karki 和 Terje 提出了疯狂的解决方法，而我成了怀疑者。

经过一段时间的每天狂热的通过电子邮件交换试验结果，我们发现 BTB 信息也许实际上是在引用分支指令之前的几个指令时就被存储了。

如果在这两者之间又来了一个分支的话，BTB 信息很可能会错误的关联到别的地方的分支。这将会导致很多有趣的现象：一个分支指令可能会拥有一个以上的 BTB entry，或者两个分支指令可能会共享同一个 BTB entry，一致于：分支会被预测到别的分支指令的目标上；一个绝对跳转指令会预测为不跳转；一个不跳转指令会被预测为跳转。

我将不描述这些怪僻的细节，但是你可以在我的网页上发现所有这些怪僻。所有的这些怪僻都不是致命的，因为所有的预测错误最后都会被修正。

一个更强大的机制

Pentium 家族的后继产品：Pentium MMX，Pentium Pro，Pentium II，Celeron，和 Xeon，都拥有更为先进的分支预测机制。我将克制讲侦探故事的欲望，而直接来谈谈这些机制。

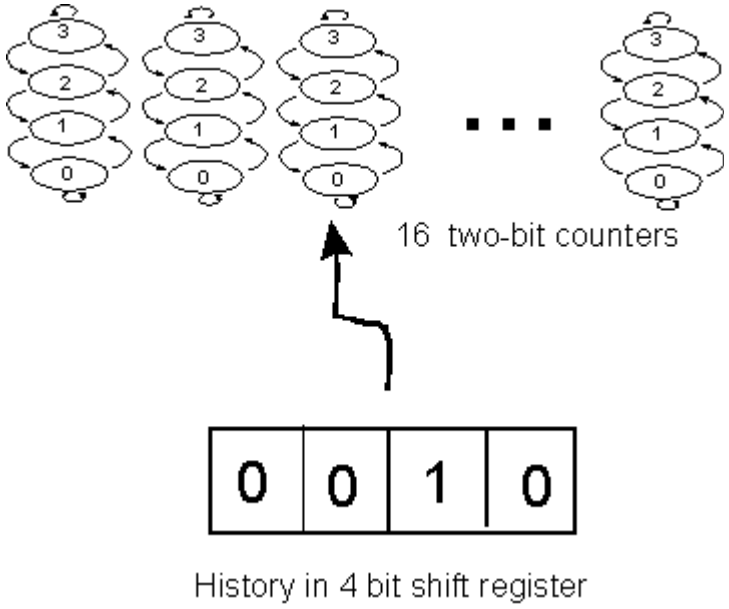
这里的机制还是基于状态图 Fig 1b 中相同的基本思想。这是一个简单的 2 位的 saturation 计数器。这个计数器在分支发生跳转的时候增加，在分支不发生跳转的时候减少。当计数器处于状态 2 或者状态 3 的时候，下一次的分支指令将被预测为跳转；当状态为 0 或 1 时预测为不跳转。这种机制确保分支不得不偏离两次，而不像大多数预测变化时那样。

在后继版本的处理器中引入了成为 2 级分支预测的改进方法。第一级是一个 shift 寄存器来存储任何分支指令的最近 4 次的事件。这就生成了 16 种可能的位样品（2 的 4 次方）。如果一个分支在最近的 4 次中都没有发生跳转那么你将获得 0000 的位样品。如果一个跳转指令在最近的 4 次都发生了跳转，那么你将得到 1111 的位样式。分支预测机制中的第二级用于生成图 fig 1b 中所示类型的 16 个 2bit 的计数器。它使用第一级的 4bit 样式去选择在第二级预测中使用 16 个计数器中的哪一个。

如图 fig 2 所示。

Figure 2

图 2　2 级 Pentium MMX，Pentium Pro，Pentium II 中的分支预测



第二级预测中保存有 16 个如 fig 1b 中所述的 2bit 的计数器，第一级是一个 4bit 的 shift 寄存器保存着最近 4 次事件的历史信息。4bit 样式是为了选择在下一次预测中使用 16 个计数器中的哪一个 2bit 计数器。

这种机制的优点在于他可以学习重新组织重复的样式。想象一个分支总在第二次的时候跳转，

你可以将这个样式写成 01010101，其中 0 代表没有跳转，1 代表发生跳转。在 0101 之后总会跟着一个 0。每一次当这种情况发生时，对应于二进制数[0101]的计数器将会减 1，直到它变为最低状态（即 0000？）。这样它就学习到在 0101 之后会跟着一个 0，并因此在下一次相似情况发生时对这个分支做出正确的预测。计数器数[1010]直到状态 3 的时候才会增加，这样它就可以总是预测在 1010 后面会有一个 1。这样就使得这个跳转的其余样式相同的情况下剩余的 14 个计数器不被使用。

这种机制比较强大，因为他可以处理类似 00101-00101-00101 这样复合的重复样式。实际上，他可以处理周期为 5 的任何重复样式，周期为 6 和 7 的大部分样式，和一部分周期高达 16 的样式。

要查看一个周期为 n 的样式是否可以被正确的预测，可以写下这个样式中的 4bit 的子序列。如果他们都是不同的，那预测机制在经过 2 个周期的初始化学习后，将可以无误的进行预测。

但是 2 级预测机制要比上述的强大的多。它还可以几乎很好的处理一个规则样式的偶然“跳转背离（deviations）”。当一个分支指令有一个几乎规则的跳转样式（除了偶然跳转背离），那么处理器将很快学会这种跳转背离的特征，然后再以后再发生跳转背离的时候就可以进行正确的预测了（这只会在第一次学习时引起一次预测错误）。

进一步的说，当两个不同的重复样式的交替出现时他也可以处理。假设你已经给了处理器一个重复样式并且处理器已经学会对此样式进行正确的预测了。然后是另外一个样式。然后是

第一个样式。如果这两个样式没有通常　的4bit的子序列，那么他们就不会使用同样的计数器，所以处理器不会去重新学习第一个样式（查找不同的计数器即可）。因此，处理器可以在引起最少的预测错误的前提下在两个样式之间前后转移。

小结

Pentium家族的第一个处理器引入了一种简单的带有不少滑稽怪癖的1级分支预测机制。之后的版本，Pentium MMX， Pentium Pro，Pentium II等等，使用了更长的管线，因此对于高效分支预测有着更高的需求。

这个需求通过强大的2级分支预测机制得到了满足，这个2级分支预测机制具有自学习、重新组织重复样式和处理规则样式的跳转背离的功能。从芯片面积的角度考虑，由于这种机制只需要使用32bit来存储一个分支的历史信息，所以也是很经济的。

2级分支预测机制的最大的缺点是在处理一个循环控制的分支样式时，预测的并不是很好。例如，你的程序中有一个循环总是要重复10次，然后在循环底部的控制指令将分支跳转回去9次，然后第10次不跳转而继续向下执行，这就会带来一次预测错误的消耗。

对于Pentium Pro和Pentium II，分支预测错误将会带来很大的时间消耗，所以，这里将一个循环改写为循环5次，然后每次循环中再循环2次的嵌套循环会好些，这样可以避免预测失误。