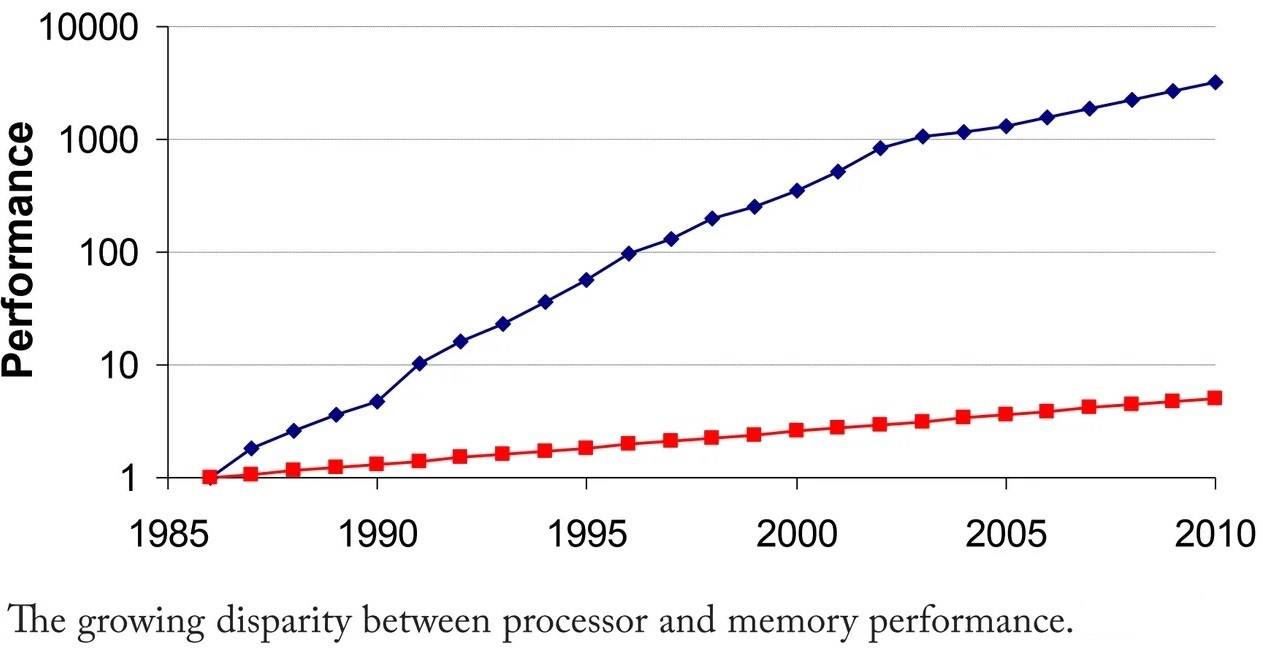
**为何需要预取**

首先讲下为什么需要预取。这是由处理器和内存间性能差异越来越大导致的。从下图可以看出，1985年至2010年间，CPU的性能提升数千倍，可是内存相关的性能只提升了不到10倍。如果等CPU需要执行相关指令或者需要修改数据的时候再从内存中去读取，那么大部分时间都会花费在等待数据上，这是不可容忍的。这一现象也被称作“内存墙”。

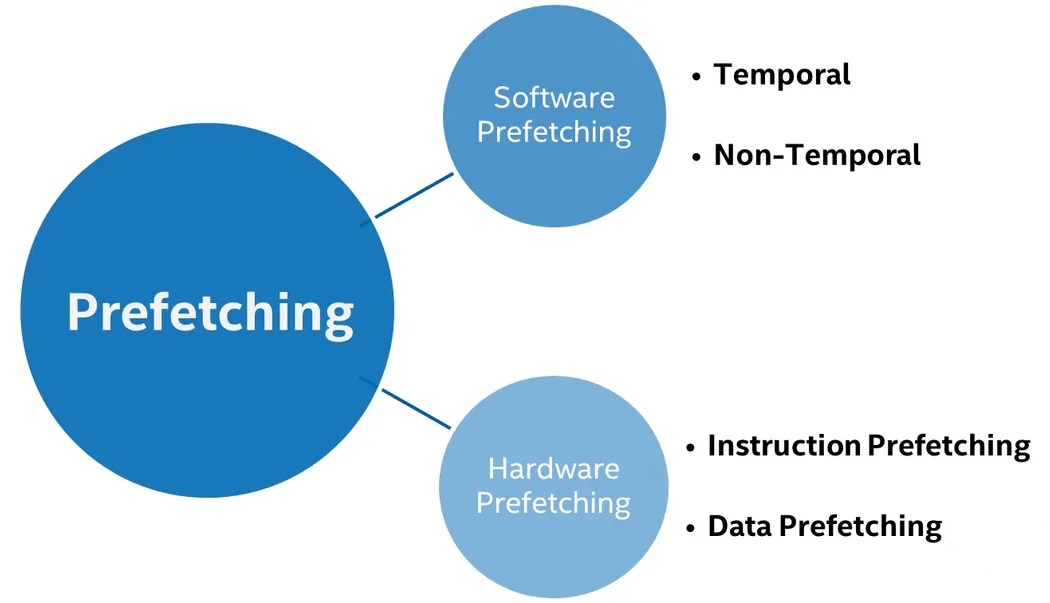


现代处理器通过多级缓存的内存层次结构，依赖内存访问的局部性来减少可观察到的内存访问时间，但不同级别的缓存对应的访问时间不同，离处理器越远的缓存所需的访问时间越长。当某一级缓存未命中时，从下一级缓存或内存获取缓存行的代价还是很大。这时预取的重要性就体现了，将将要访问的内容提前从内存搬移到Cache中，CPU就可以即时拿到所需的内容，避免了等待。

**预取机制的分类**

预取的机制可以分为软件预取和硬件预取。软件预取的前提是处理器支持预取指令，可以是在编程时主动使用预取指令来执行预取，或者是在编译时使用优化的编译选项，让编译器加入预取指令以提升程序的执行效率。

硬件预取是由CPU内部专用的硬件模块来实现的，CPU内部的预取机制会一直观察分析所执行的指令或数据流，从中找出规律，从而提前将可能要被用到的数据放在相关的缓存中，以待被使用。硬件预取又分为指令预取和数据预取，我们组重点讲解数据预取技术。



**硬件预取的指标**

CPU内部自动实现的硬件预取是CPU架构学术界研究比较多的领域，并且对CPU的整体性能影响也很大。

硬件预取策略需要考虑以下三点：

1. 预取哪块数据：必须能准确地判断所需预取的数据，无效数据对我们毫无用处，只会白白耗费电力；
2. 何时开始预取：如果预取不及时，甚至晚于需要相关数据的节点，那么这个策略毫无帮助，浪费硬件资源，但若过早，预取的块还没有被使用可能就被再次替换了；
3. 数据存放在哪：如果相关预取数据存放不合理，将会将后续需要用到的数据踢出CPU，反而可能会造成性能下降。

在评价一个预取机制策略的好坏时，可以从覆盖率和准确率两个关键指标来评判：

覆盖率是指预取所消除缓存未命中（Cache Miss）的比例。比如没有预取机制的时候会发生100次Cache Miss，引入预取将Cache Miss减少到了30次，意味着该预取策略成功避免了70次Cache Miss，即覆盖率为70/100=70%。

预取的准确率还需要考虑在这个过程中预取机制发出了几次预取操作。接着上面的例子，如果该预取机制总共发出了140次的预取操作，其中70次是有效的预取操作，那么准确率为70/140=50%。

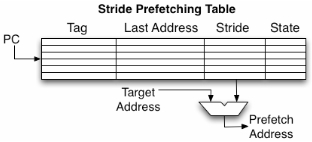
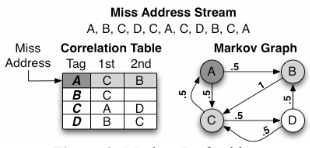
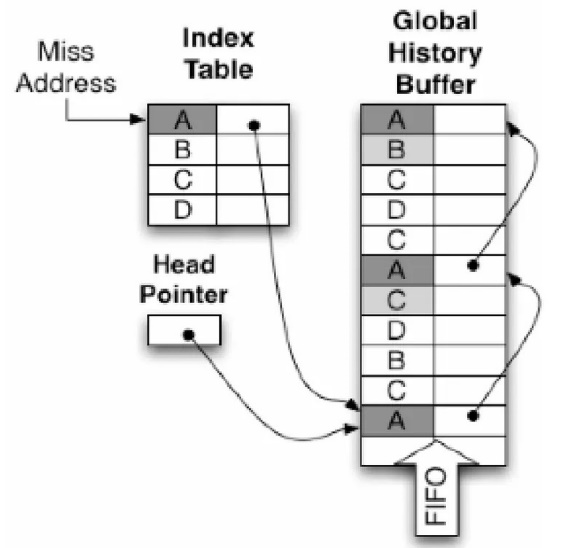
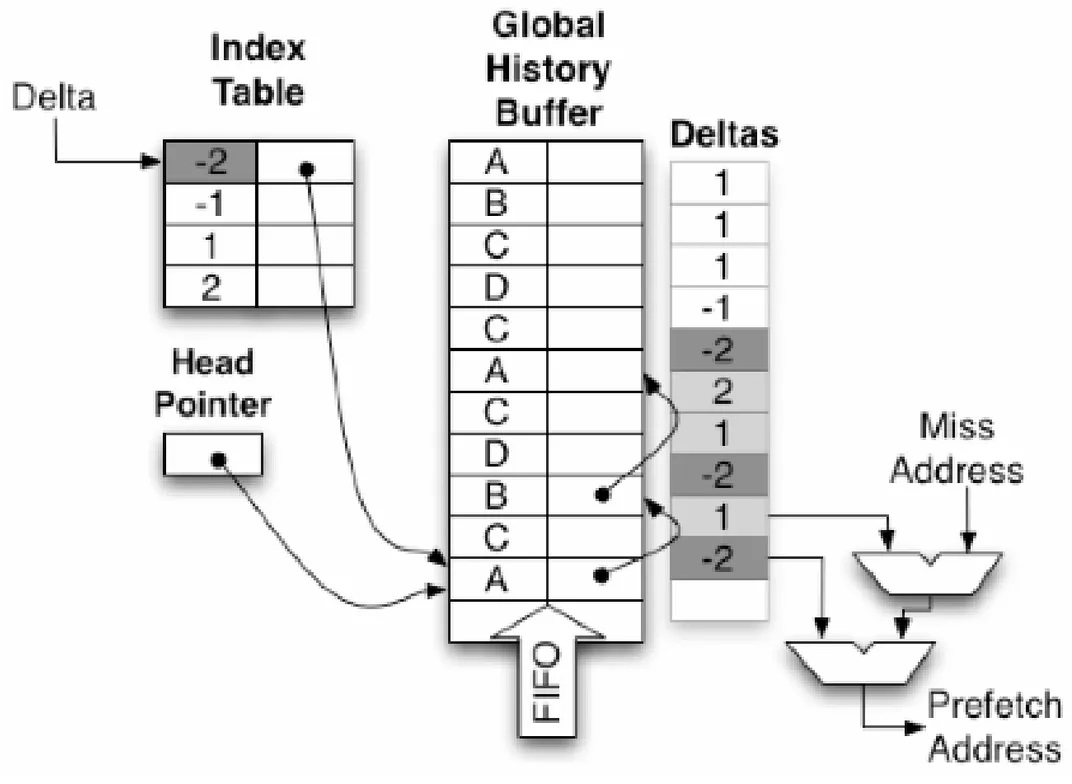
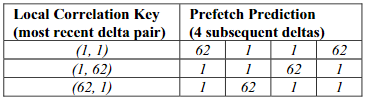
在不同时考虑两个指标的情况下，单纯盲目地提升覆盖率指标是很容易做到的，宁可错杀一千，不可放过一个，这种情况下覆盖率大大提升，但是准确率会极速下降，最终性能不升反降。理想的预取策略应该在取得高覆盖率的同时，确保准确性。

除了覆盖率和准确率，时效性也是预取很重要的一个衡量标准。

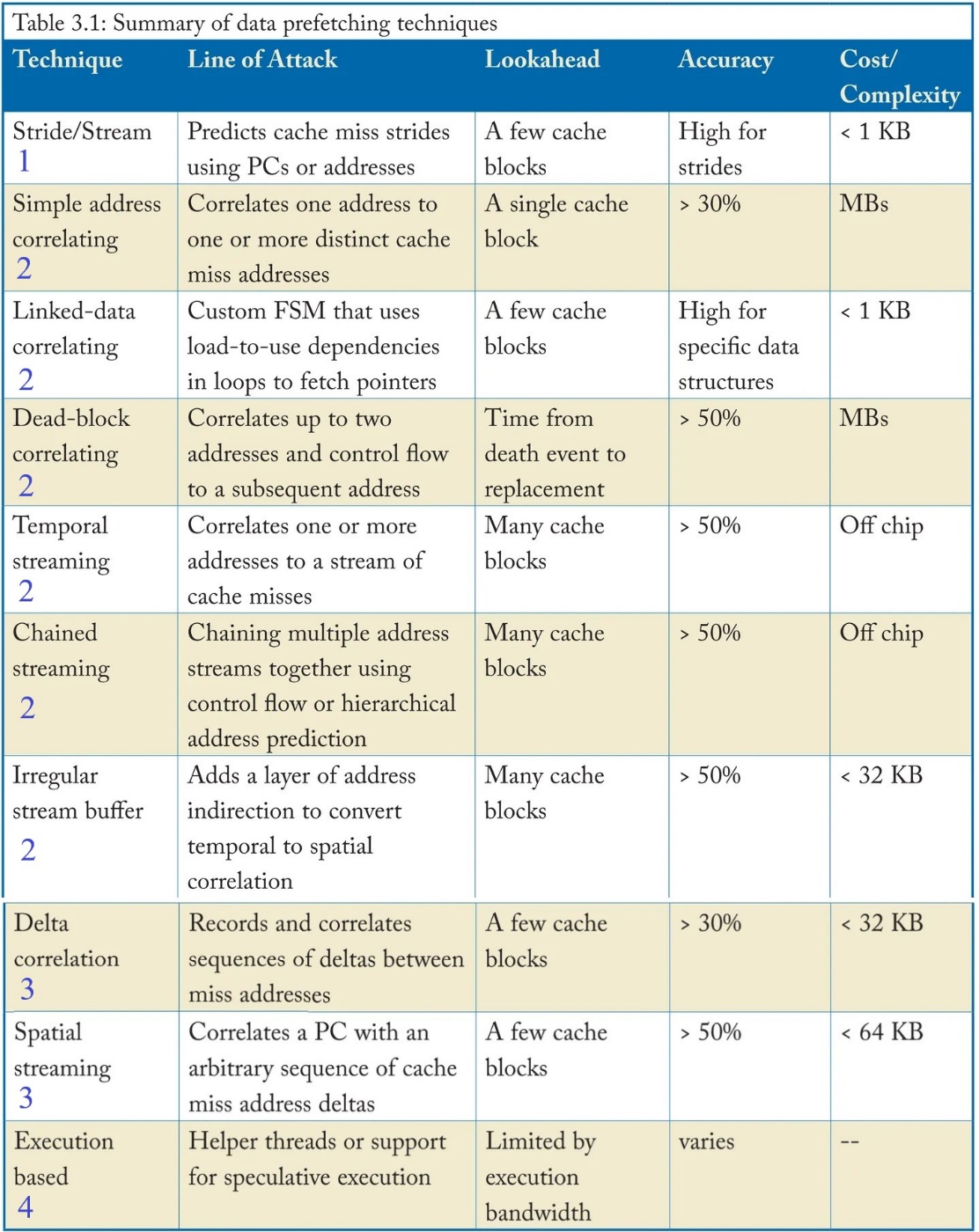
**数据预取的类型**

接下来具体讲解数据预取技术。由于数据访问的模式相对指令来说更加复杂多变，尤其是还会有很多遍历执行和通过指令链接的数据结构。当然，同时这也意味着有更大更丰富多彩的设计空间，设计人员可以天马行空地设计他们的数据预取机制，只要最终的结果让人满意。

我们大致可以将数据预取简单分为四类：

1. Stride and stream prefetcher：简单根据缓存行（Cache Line）的访问步幅（Stride）来进行下一个Cache Line的预取，类似指令预取中的Next-line预取，跨步幅访问的长序列称为流（stream）。  
   该机制最简单的实现方式是，记录每次load或store指令访问的地址，并且还会记录最近两次访问地址的间距。只要该模块连续两次观察到相同的间距时，便会用最后一次访问的地址加上相应的步幅作为地址进行提前预取。这种机制对于数组和密集矩阵访问十分有效。  
     
   当然，在实现时还有更多问题需要考虑。比如，需要不断检查步幅来动态调整置信度，当置信度减为零，代表数据流已经不连续或者是连续的步幅已经不等于之前计算得到的步幅，按照之前的步幅进行预取已经没有意义，反而有可能会污染缓存，因此此时需要停止预取。  
   又如，在进行矩阵向量乘法、多个矩阵交替访问时，每个矩阵访问的步幅可能不同。对于这种交错跨步幅的序列，可以转而记录重复执行的同一条load/store指令（相同PC）的访问地址间的步幅以进行预取。  
   这种方式也被称作多流预取模式，而前一种可称为全局预取模式，这两种模式就是玄铁C910数据预取的主要实现方式。
2. Address-correlated prefetching：针对重复遍历的序列进行记录，在发现重复模式时进行预取，这种对指针的数据比较有效。  
   该机制的典型实现是马尔科夫预取器，它会记录未命中的数据地址后续未命中的地址，一般会保存四个候选地址。在访问某个地址时，在相应的表格中查找，如果找到，可以将候选地址中的全部取出，也可以做出调整，只取出访问频率最高的那一个。该策略等于维护了一个地址访问记录日志，把某个地址后续可能会访问的地址记录下来，而出于资源的限制，只能记录少数的地址和少数的候选项。（命名来源：马尔科夫概率模型）  
     
   这类预取模式还有许多其他实现机制，比如距离预取，采用两个连续的未命中地址的间距作为索引。出于时间原因不再一一介绍，后面会有汇总表格。
3. Spatially correlated prefetching：针对常规的数据结构进行针对性地设计相关预取策略。  
   空间相关性预取机制利用数据访问的规律性和重复性，这种机制对面向对象编程中的对象，固定大小字段的数据库数据等有规律的数据块进行访问时比较有效。目前许多高级编程语言会将数据结构与Cache Line和页边界进行对齐，以提升其数据访问的规律性，更容易发挥这种预取机制的潜力。  
   从存储效率和覆盖范围的角度来看，最有效的预取器之一是全局历史缓冲区（Global History Buffer）的局部PC增量相关变体（GHB PC/DC）。  
   简单来讲，GHB预取器利用未命中地址在索引表中查找得到GHB的索引，即索引表中的表项包含指向GHB的指针，GHB 表项代表待预取的流。这将表键匹配与预取相关的历史信息的存储解耦，以此加深预取深度。此外，GHB项中还可以包含指向另一个项的指针，从而扩大预取宽度。这有效解决了马尔科夫预取器的深度太大则存储效率低，深度太小则lookahead不够的矛盾。  
     
   基于GHB，有地址相关（Address Correlating (AC)）和增量相关（Delta Correlating (DC)）两种预取机制。前者直接用未命中地址索引, 得到GHB的序列开始位置。后者当发生未命中时，在索引表中找出该PC对应的GHB指针，并回溯GHB中前两个与该PC相关的项，得到其访问地址的步幅模式，随后继续回溯GHB中与该PC相关的项，直到匹配到相同的模式。此时我们可以认为找到了上一次访问的数据访问模式，并利用这个信息完成预取，这种机制效率更佳。（图为全局增量相关）  
     
   后者，也被称作全局增量相关（G/DC），还有一种改进——局部PC增量相关（PC/DC），它使用增量对(两个连续的增量)作为相关键（correlation key）。  
     
   
4. Execution-based prefetching：在CPU的乱序阶段前就进行预取相应的识别和预取，不依赖内存地址访问的规律性和重复性。  
   这种策略利用CPU中的Stall周期和空闲的硬件资源，提前在执行之前把指令地址提前在Cache中准备好，有点提前找到考试内容并打小抄的感觉。

下面是各种预取机制的总结表格。



**参考**

[1]【预取简介】[Prefetching Introduction]-拓荒犬的文章-知乎  
<https://zhuanlan.zhihu.com/p/491673969>；

[2] A Primer on Hardware Prefetching (Springer Nature)；

[3] A Primer on Hardware Prefetching - winfred的文章 - 知乎  
<https://zhuanlan.zhihu.com/p/150244427>；

[4] K. J. Nesbit and J. E. Smith, "Data Cache Prefetching Using a Global History Buffer," 10th International Symposium on High Performance Computer Architecture (HPCA'04), Madrid, Spain, 2004, pp. 96-96, doi: 10.1109/HPCA.2004.10030.

By 22307140106 金宇坤