# 预取介绍

Cache是一种利用程序访问局部性原理所设计的存储机制，其目的就是尽可能弥补处理器计算速度和存储访问速度之间的巨大鸿沟。提高存储器访问性能的方式之一是降低cache访问的miss ratio。预取就是一种通过提前将可能需要的cache line取来从而降低Compulsory miss的方式。预取本质上是一种对程序存储器访问地址的猜测机制，所以需要对程序数据结构特性的分析，其性能提升也与所执行的程序特性表现出较大的相关性。但是和分支预测不同的是，猜测错误的预取操作不需要进行恢复。

# 预取机制

预取是要提前把一个或者一些cache line取来，有三个问题需要考虑：什么时候取？取哪些块？取来放在哪里（如果直接放在cache里要设置怎样的替换状态）？所有的预取机制的实现都是对这三个问题的回答。针对不同的程序结构，不同的目标cache，采用的预取策略不同。预取很容易造成cache污染，实现需要谨慎，做的不好可能不仅不会提升处理器整体的性能，还会降低。

预取不能造成正常程序存储器访问阻塞，有三种指导思想：给cache增加访问端口；存储器空闲周期再发起预取访问；记录预取地址从而避免不必要的重复访问。

预取需要解决的一些问题：

1. 取到错误的块----优化预取算法
2. 预取不及时----预取间距的设置
3. Cache 污染----增加临时buffer

# 预取方法

## Always prefetch (1982)

每次存储访问（normal reference）第i个cache line都会检查第i+1个line是否在cache中（prefetch lookup），如果不在，发起预取，正常的存储访问，预取查询，以及预取来的块都会更新LRU的状态，三种访问不予区分（因为Smit78b文献中表明区分三种访问对LRU状态的更新并没有什么明显的性能收益）。

Always prefetch可以减少75%-80%的miss ratio，但是额外增加20%-80%的块搬移。

## Prefetch on miss (1982)

Prefetch on miss是指在访问第i个cache line时如果发生cache miss，除了将miss的块取来，还会发起对第i+1个块的预取。

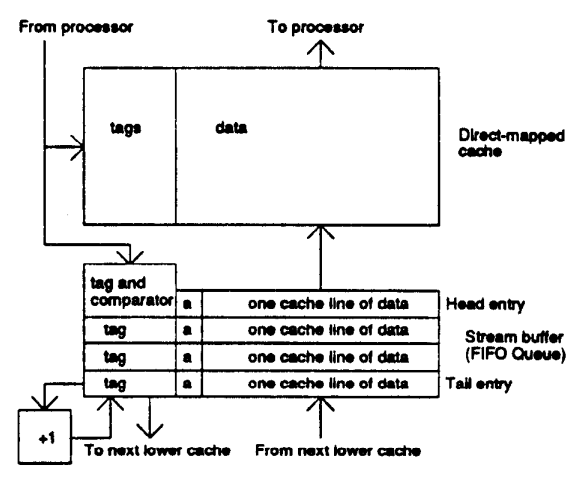
Prefetch on miss所减少的miss ratio是always prefetch的一半甚至更少，但只额外增加了10%-20%的块搬移。

## Tagged prefetch (1982)

Tagged prefetch会为cache中的每个块设置1bit的tag标识位，该值被初始化为0，块被miss取来或者正常的访问时会将0变为1，每次tag从0到1的变化都会触发对下一个顺序块的预取，预取来的块tag仍然为0，当块被搬出cache时tag会被重新置为0。

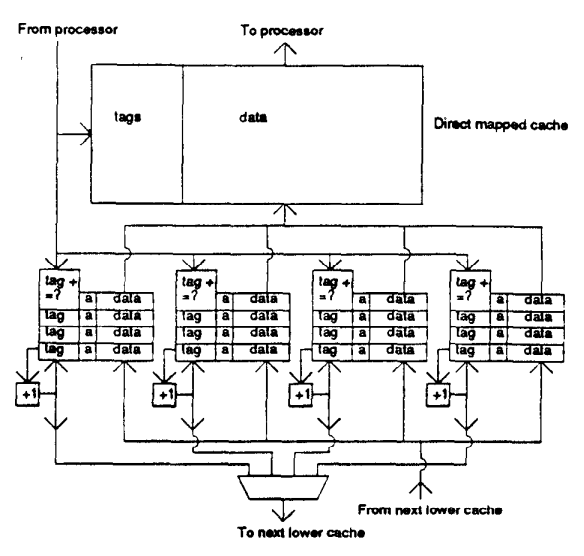
Tagged prefetch所减少的miss ratio和always prefeth相当，但是增加的块搬移率和prefetch on miss相当。理想情况下，对于纯粹的顺序流访问，tagged prefetch可以将miss ratio减小为0，需要解决的问题是块预取速度跟不上真正的块使用速度。

## Stream buffer (1990)



Stream buffer是一个小容量的顺序块FIFO，每一项的内容和一个cache line类似，有tag地址字段，数据字段，以及块数据是否可用的a（available）字段。当cache访问发生miss的时候，miss块之后的块会被依次取来放入buffer中，数据已经完全从下一级存储搬来的块的a字段会被设为1，tag字段则用来计算下一个要搬移块的地址。Stream buffer会像流水一样将一连串的块搬入buffer直到buffer满或者遇到其他不必再进行数据搬移的情况（比如到达页边距）。这样子当一个块miss的时候，stream buffer就已经把后续块都准备好了，如果程序保持顺序访问，下一个发生miss的块就可以快速地从buffer中搬到cache里，不再需要耗费大量时间访问下一级cache来进行refill操作。Stream buffer严格遵循顺序流，当miss发生的时候只看FIFO的第一个块是否是miss块，如果不是就仍然发起下一级cache的refill操作，同时stream buffer也会被清空并发起新一轮的流缓存操作。Buffer项数的设置取决于执行程序所能维持的顺序块数目。

Jean-Loup的文献数据显示，Stream buffer技术可以消除72%的指令cache miss，但是只能消除25%的数据cache miss。这是因为指令流很大程度上都是顺序流，但是数据流则会至少在两个流之间切换（比如字符串比较），频繁的流切换会使stream buffer失去其效力。所以前人在stream buffer的基础上提出了multi-way stream buffer来进一步提升stream buffer技术对数据cache的效力。



Multi-way stream buffer拥有多个流缓存buffer，新数据流的miss发生时不必清空当前stream buffer，换下一路buffer来缓存新的数据流。判断miss地址是否在buffer中有缓存时需要比较所有路的buffer头块的地址，当所有路都已满且不命中时才需要清空其中一路来进行新的流缓存操作，一般使用LRU算法来选择要被清空的路。

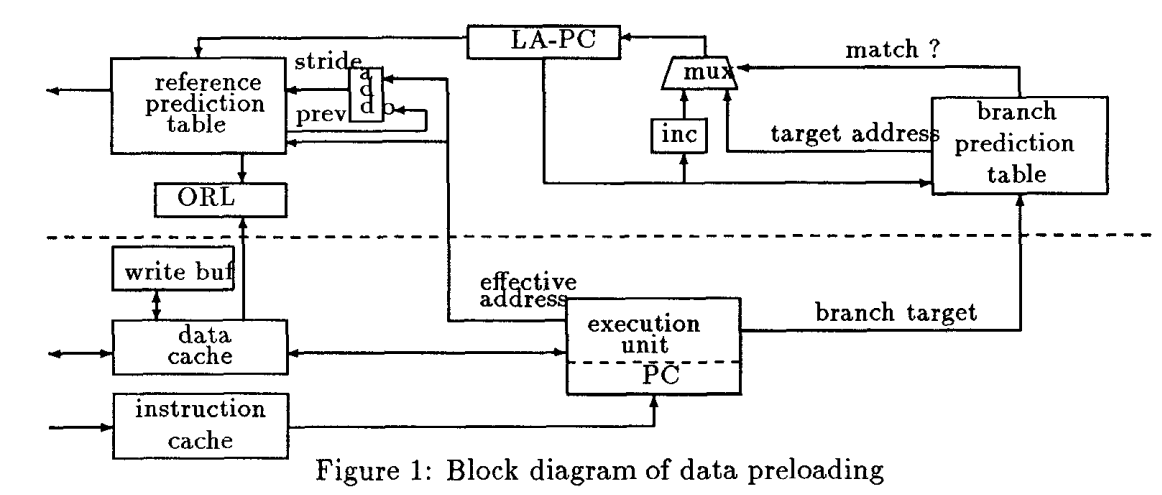
单路的stream buffer对指令cache的效力几乎已经发挥到极限，所以multi-way stream buffer对指令cache的进一步提升空间有限，但是它对数据cache miss的消除力度达到43%。

对于要写回的块，若块也存在buffer中，数据也要被bypass给buffer中的块。

## Stride direct prefetch (1991)

Stream buffer技术对于ICache这种大多数情况都是顺序流访问的性能提升很明显，但是数据访问的模式更多变，所以即使是multi-way stream buffer对数据 cache的性能提升也很有限。所以前人提出了针对DCache的预取策略，其主要探索的循环体内访存指令的数据流。这种方式对于数据访问模式比较规整的科学计算程序性能提升尤为明显。

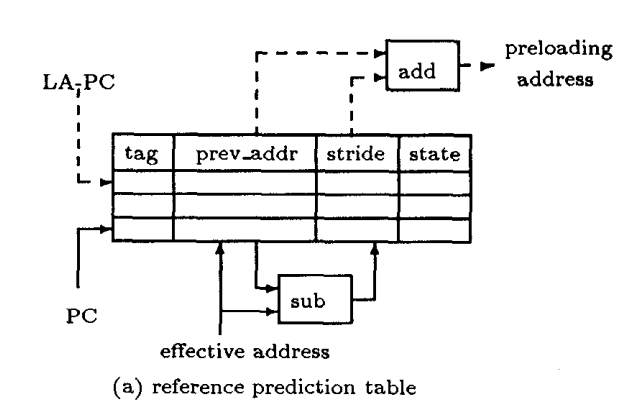
文献将存储器访问分为四种模式：scalar（比如循环控制变量），zero stride（比如多层嵌套循环中外层循环的某个变量在当前循环中保持不变）， constant stride（比如数组元素地址的变化），irregular（不属于上述情况的任何一种）。作者提出了一种LA-PC(look-ahead PC)+RPT(Reference Prediction Table)的预取策略来探索存储指令的数据流模式并在取值时根据分支预测的指令地址提前对后续可能访问的数据地址发起预取操作。



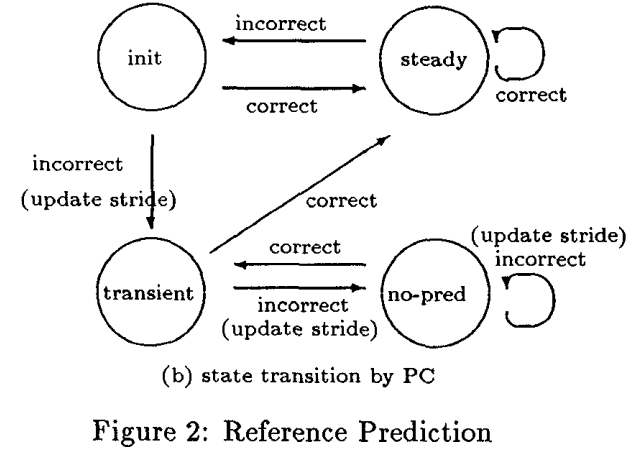
大致的数据预取结构如上图所示：RPT中记录的存储器访问指令的访问模式，由实际执行后的访存指令对其内容进行更新修改，用BPT(Branch Prediction Table)预测出的LA-PC来访问RPT，如果RPT不命中，不进行任何操作，如果RPT命中则证明是当前被使用的一个数据流，硬件会检测是否是一个regular的数据模式，将被使用的数据块是否已经在cache里，通过检查ORL判断（Outstanding Request List，预取请求列表）是否已经发起过对相应数据块的预取操作，如果都没有，则发起一个预取请求操作，该请求会被放入ORL中，预取来的数据会被直接放入DCache中。这样如果预取猜测正确，且数据搬移及时，在指令发起真正的数据访问操作时，数据就已经在DCache中了。当发生一个DCache miss时，会率先检测ORL中是否有相应数据块的预取请求，如果有，等待直到数据被搬移到cache中；如果没有，发起一个优先级比所有ORL都高的miss request操作。如果发生分支预测错误，ORL中的所有预取请求都会被清除。如果预取请求导致了异常，比如page fault，预取请求会被忽略。

接下来对RPT的结构进行说明。如下图所示，它是一个直接映射的cache结构，每一项包含tag, prev\_addr, stride, state字段，使用LA-PC进行访问，使用指令执行后的PC后有效地址对其进行更新。Tag是部分的指令地址字段，prev\_addr是访存指令上一次存储器访问的有效地址，stride字段记录的访存指令访问的步长， state字段记录的是对当前数据流模式学习的程度以及是否发起预取操作。

LA-PC只查询使用RPT，而整个RPT的内容是根据访存指令执行后的PC和有效地址进行更新的。



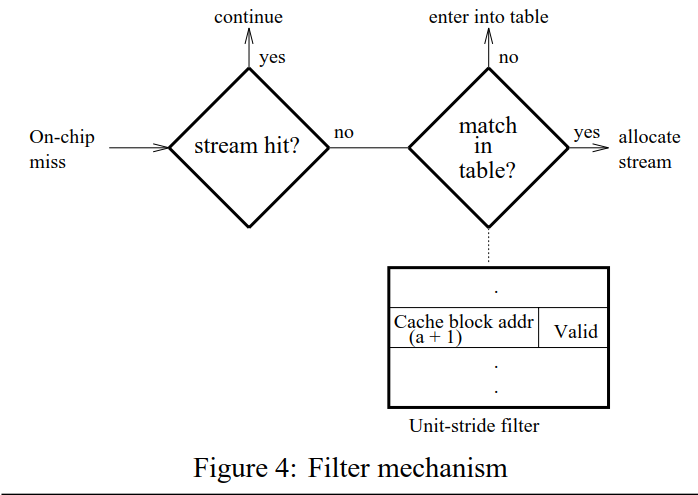
下图是RPT中状态机的示意图。每次访存指令计算出有效地址后都会根据PC去查RPT如果RPT命中，会判断当前有效地址是都等于prev\_addr+stride，如果相当则认为RPT所给预测信息正确，并将新的有效地址存入pre\_addr字段，sate会给出steady状态；如果不相等，则认为RPT所给预测信息错误，会将有效地址存入prev\_addr字段，并将有效地址减去prev\_addr存入stride字段，状态机也根据图示做出相应的状态转换；如果RPT不命中，则会将指令PC存入tag字段，有效地址存入prev\_addr字段，stride设为0，state设为init状态。



文献使用CPI对预取机制的性能进行的评估，数据显示对于N KB的DCache，加入256项RPT和256项BPT的预测机制可以将CPI减少10%-95%。

## Unit filter stream buffer (1994)

纯粹的stream buffer技术会发起对miss地址之后大量顺序块的预取操作，所以会极大增大存储带宽的压力。于是[Palacharla 1994]提出了对miss地址的filter机制，连续两次对同一个流的miss访问才会发起对相应流的预取操作，从而过滤掉那些只有一次访问操作导致的不必要的流预取操作，一定程度上提高了预取的精度。



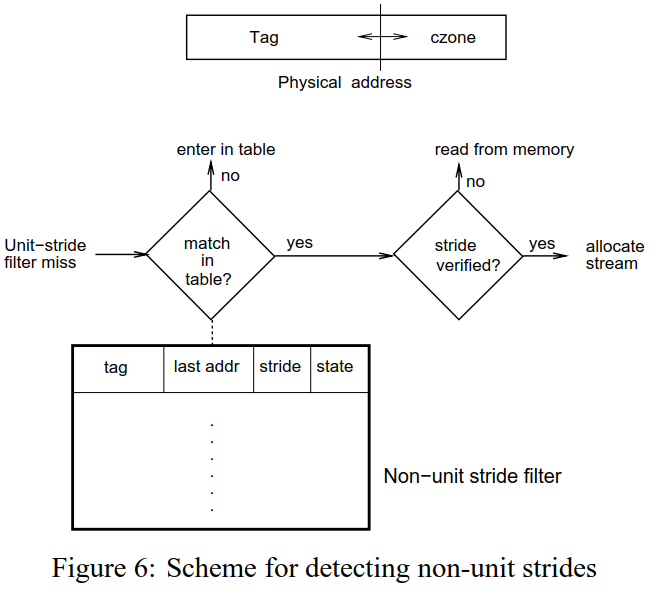
这种过滤机制的工作原理如上图所示，在原有stream buffer的基础上增加了一个miss 地址的历史记录表，叫做unit-stride filter，深度为8到10项，每一项存储的是miss块地址加1的地址，同时有相应的valid标识位，整个表采用全相连方式的搜索，当表项满时会将旧地址替换出去。

当有一个地址miss发生的时候，会首先查找stream buffer查看是否hit，如果hit直接将请求数据块从stream buffer搬移到cache中，如果stream buffer不命中，则miss地址会送到filter检查是否之前已经发生过同一个数据流上的miss，如果命中，则说明已经连续发生了两次同一个数据流上的miss，那么很可能这个数据流仍然会被程序所使用，此时发起对miss地址下一个数据块的预取操作，并且filter中的相应miss地址记录会被释放。如果miss地址在filter中没有命中，则将miss地址+1的块地址存入filter中。

文献数据显示，对于大多数的benchmark, unit-stride filter能减少50%的额外带宽需求，而cache的命中率几乎不变。具体测试数据参考性能参数对应小节。

## Non-unit filter stream buffer (1994)

对benchmarks的进一步的数据访问特性分析显示，有些程序存在大量的非单位步长的存储器数据访问，之前提到的non-unit stride filter无法检测到这样的数据流，所以预取对这类程序的性能提升有限。于是，[Palacharla 1994]在同一篇文献中提出在unit-stride filter的基础上增加non-unit stride filter，可以按照一定的步长来检测数据访问模式，从而发起对后续访问地址的预取操作。这样的两级miss地址过滤器，可以进一步提高预取精度。



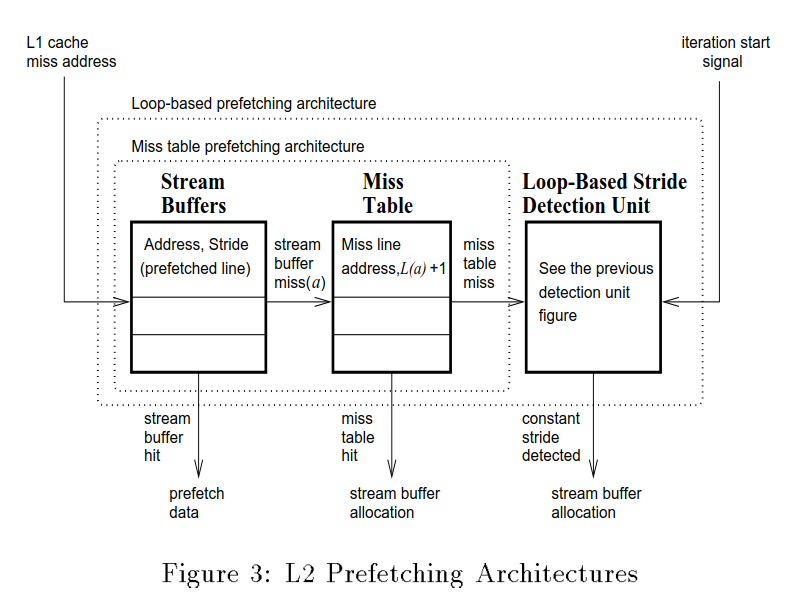
Non-unit stride filter是一个和RPT类似的访问模式记录表，只是之前提到的流定位使用的是访存指令的PC，而因为当前文献所提出的预取机制应用于统一的二级cache的预取，所以流定位使用的是物理地址。所以通过将物理地址空间分区（zone）的方式来区分流，落在同一个区内的地址属于同一个流，然后在同一个区内探索miss地址之间的stride关系，当新的miss地址和前两次落在同一个区的miss地址保持相同的步长，则发起对miss地址加步长地址的预取。同样，一旦发起对特定流的预取操作，相应的filter记录项会被释放。

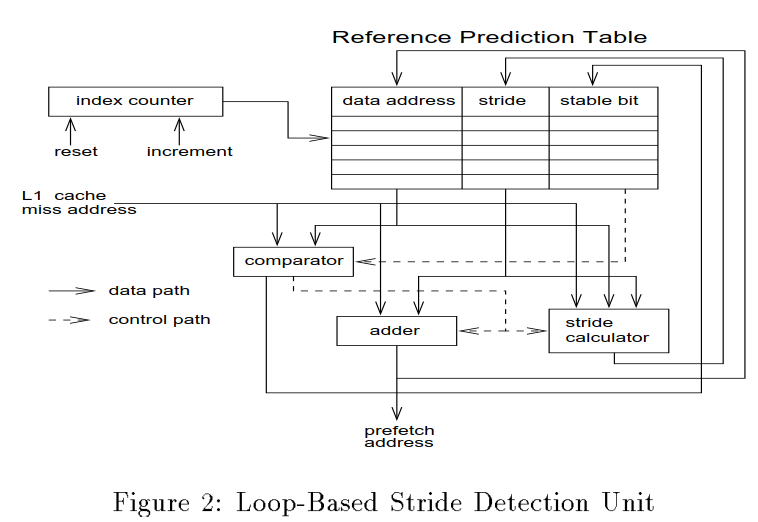
文献数据显示这种non-unit stride filter可以大大提高有大量非单位步长benchmarks的性能。具体测试数据参考性能参数对应小节。

## Loop-based prefetching (1997)

对于指令和数据统一的二级cache，往往采用的是unit-stride filter + non-unit stride filter的stream buffer技术，unit stride的顺序流检测比较简单，使用块地址加1即可，但是non-unit stride的stride检测则相对比较困难。对于一级DCache的预取可以使用访存指令的PC来检测数据流的步长，但是对于二级cache，所能使用的只有数据访问的物理地址，所以基于PC的流检测技术不再适用。已经提出的二级cache的流步长的检测技术有物理空间分区技术（CZone）和最小间距技术（minimal delta），而[Sunil Kim，1997]提出一种新的L2 Cache的流检测方式：loop-based prefetching。

下图是loop-based prefetching所处位置，和[Palacharla 1994]提出的L2 Cache预取器结构一致，采用Loop-base stride detection unit做为non-unit stride filter。





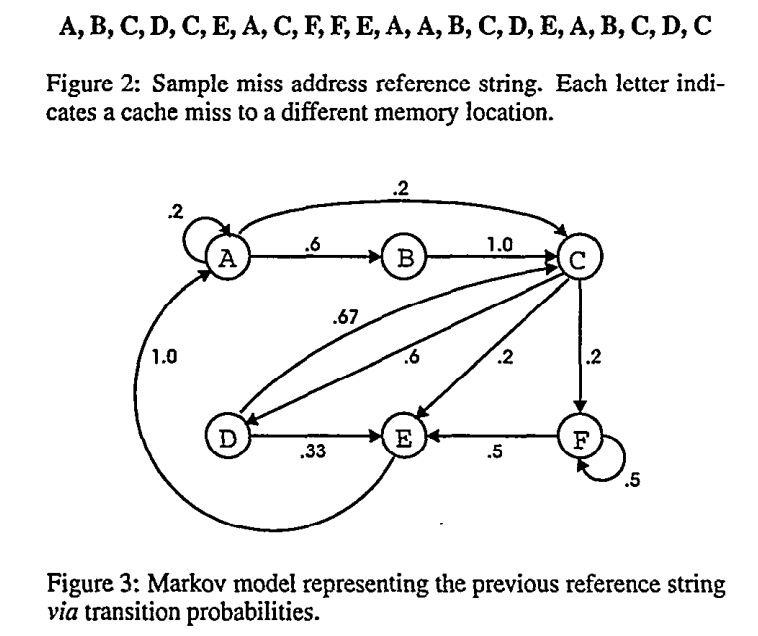
Loop-based stride detection unit的结构如上图所示，其中RPT指针不是使用PC，也不是物理地址，而是一个指针计数器。循环体中的访存指令会被重复执行，新一轮循环中的访存指令的执行顺序往往和上一轮循环的执行顺序保持一致。如果能获取每一轮循环的开始标识（比如特定的访存地址，比如分支的回跳），就能通过访存指令距离循环开始标识的距离来区分存储访问流，这个相对距离作为RPT的访问指针，每一轮新的循环都会将指针counter置0。如果指针所指项有预取请求，则会在L1请求前发起预取操作。如果计数器溢出，即循环体访存指令数目超出RPT项数，所有后续访问会被忽略。

文献数据表明，这种预取策略因为使用了cross-stream locality，其对L2 Cache的预取精度反而要比PC索引的RPT预取效力更好一些。

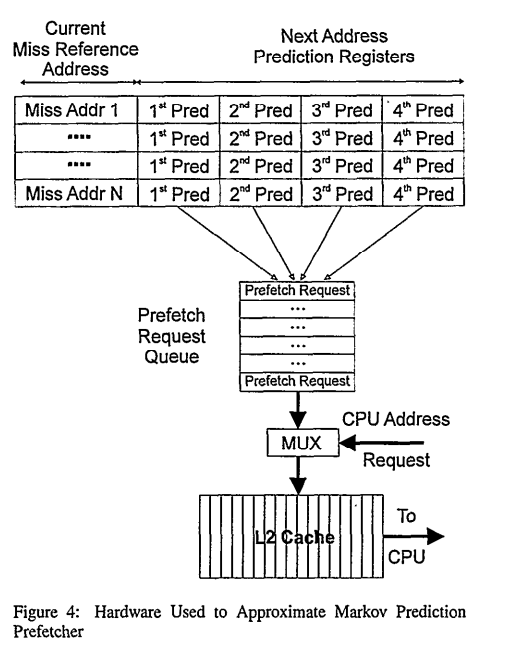
## Markov Predictor (1997)

之前提出的预取技术都是基于对测试程序比较规整数据结构访问模式的探索，但是有很多技术和商业应用会使用大量的指针结构，数据访问地址的规律性不容易探索。对此，有人提出了关联预取的方法，一种通过发生miss地址之间关系的概率分析模型来构建的预取机制。[Joseph, 1997]年提出的Markov预测器就是一种关联预测器。

我们简单介绍一下Markov对数据地址访问模式的探索。下图有一个用A,B,C,D,E,F表示的miss地址串，通过相邻两个地址间关系的探索，我们可以将这个访问模式串转化为一个关系图。从图中可以看出，当一个地址miss发生的时候（比如A节点），紧接着可能的miss地址和其发生的概率其实是可以推测出来的，比如A之后可能的miss地址有A，B，C，其中A到B发生的概率是A到C的两倍因此我们在看到A地址的miss后可以发起对B和C的预取。



Markov预取器就是硬件记录并检测以上数据模式的过程。因为硬件资源的有限性，纯粹的Markov模型不可能被实现，我们只能对有限的miss地址进行记录，每个miss地址之后的可能跳转地址的记录数目也是有限的，向每个地址跳变的可能性计算也不能采用纯粹的概率值。下图是一个Markov的硬件实现示意图。



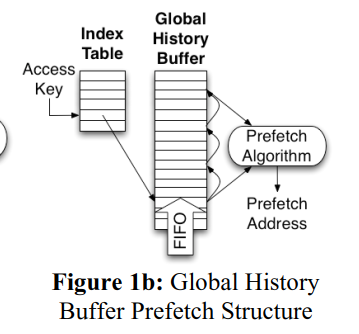
硬件实现中，有一个表格对Markov模式转换图进行记录，每个节点是表格中的一项，并且每个节点最多拥有4个可能的转换节点，每个节点的优先级设置不同。当一个miss地址在预测表中命中的时候，会发起所记录四个后续地址的预取请求，预取请求会被放在预取请求队列中，不同地址拥有不同的优先级，当预取请求队列满的时候高优先级的请求会将低优先级的请求替换出去。一旦一个预取请求被L2 cache响应，相应的数据会被存在一个片上的全相联的buffer中，buffer是一个FIFO结构。当L1 cache发生miss的时候首先会搜索整个buffer，命中的话就直接使用数据，不命中才会请求L2 Cache。

周期级的模拟器测试数据显示，对于不同的商用测试程序，Markov预测器只使用原纯粹cache结构存储资源的2/3可以将MCPI平均减少54%。

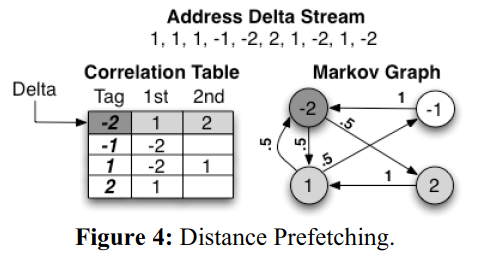
## Global History Buffer (2004)

在[Nesbit, 2004]提出GHB（Global History Buffer）之前，所有的预取策略都采用的是直接索引的表格来记录历史信息，这样的存储结构存在几个问题：一，有些项的内容可能会老化但并不能被及时替换出去，这就导致预取精度的下降；二，同一个项之间的数据冲突可能导致一个活跃数据流被替换出去；三，表格每一项所能记录的历史信息数目总是有限的。

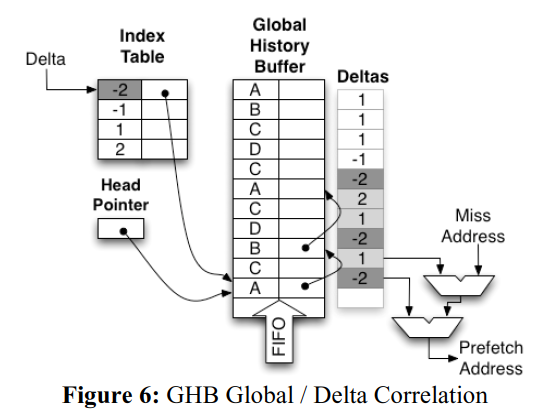
而GHB采用的是一个FIFO记录表的形式，所以的miss地址都会按照顺序存入GHB表项中，当GHB满的时候，每个新的miss地址都会将一个最老的miss地址挤出FIFO。而同一个数据流上的信息会以链表的形式关联起来，链表通过一个哈希表间接索引。下图是GHB的基本结构。它包含两部分，IT（Index Table），存放链表的首地址，当访问IT后会拿到一个GHB的地址，它是同一个数据流miss链的首地址，GHB的每一项除了miss地址外，还存储了链下一个节点的GHB地址，可以沿着链节点的索引地址一直访问下去。GHB提出的是一种链式关联的存储结构，它最大的优点就是历史信息记录总是可以被及时更新，且每个数据流上所记录的信息是由流访问的活跃程度决定的。我们可以用这种数据结构实现任意的预取机制。比如IT的索引键可以是miss地址，也可以是PC，我们可以用GHB实现stride 预取，Markov预取，甚至任何形式的关联预取。



我们举一个GHB G/DC（Global miss address/Distance Correlation）关联预取的例子。下图是传统的Distance Prefetching的示意图，当发生miss的时候用当前miss地址和上一个miss地址的差值作为索引访问关联表，关联表中存储的是历史miss地址差值序列，使用当前miss地址和历史差值进行累加来获取所有的预取地址。



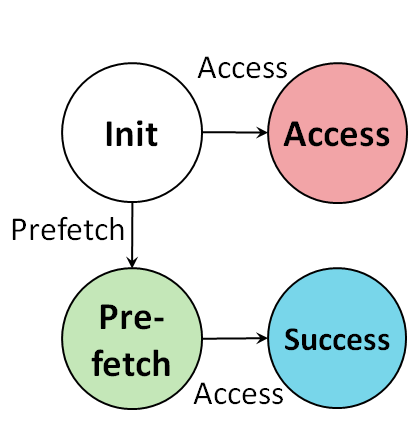
下图是用GHB的形式实现的 Delta关联预取的结构示意图，IT的索引是miss地址的差值，GHB中存储的依然是历史miss地址，地址之间的链接关系是miss差值。当用miss地址的差值索引IT后找到上一个相同差值的miss地址后，得到其后续差值后计算出预取地址。

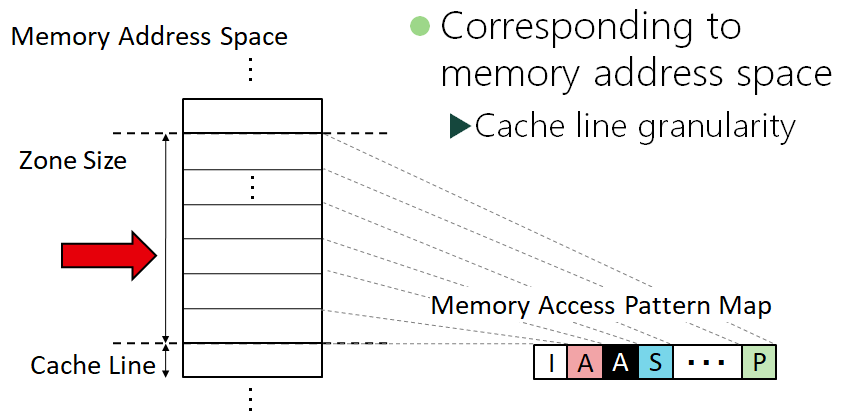


文献统计数据显示，GHB 的G/DC方式比传统的G/DC的性能提高为20% IPC。

## Access Map Pattern Matching Prefetch (2009)

[Ishii, 2009]提出的是一种全新的预取策略，应用于L2 Cache的预取。这种预取策略将整个物理地址空间分区，每个区的所有cache line的状态都会被标记起来。如下图所示，每个cache line有四种状态，Init代表初始状态，此时cache line不在cache中，Access代表被miss访问填到cache中的line， Prefetch代表被预取请求填充到cache中的line。Success标识被预取请求填充且被访问了的cache line。会有一个N项的存储器映射表，可以记录最近的N个区域内cache line的状态。当cache miss发生的时候，使用miss地址访问存储器映射表，取出miss地址所在区所有cache line的状态信息，然后通过一定的逻辑计算得出可能的预取地址。这种预取机制最大的优点是它的历史信息记录完全不依赖于访存操作的顺序，且有很强的自使用能力，对硬件的优化以及编译器的优化都很友好。





文献数据显示，这种AMPM预取器能够将性能提升53%，并且能减少74%的L2 Cache miss.

# 性能参数

# 商业实现

# 实施方案

Icache

DCache

L2-Cache

1982：Cache Memory, A. J. Smith

1990：An Effective On-chip Preloading scheme to reduce data access penalty, Jean-Loup

1991：Stride Directed Prefetching in scalar processor, Edward H. Gornish

1994：Evaluating stream buffers as a secondary cache replacement, Subbrao Palacharla

1997：Stride Directed Prefetching for Secondary cache, Sunil Kim

1997：Prefetching Using Markov Predictors, Doug Joseph

2004：Data Cache Prefetching Using a Global History Buffer, Kyle J. Nebist

2005：On the importance of optimizing the configuration of stream prefetchers, Llya Ganusov

2009：Access Map Pattern Matching Prefetch: Optimization Friendly Method, Yasuo Ishii