降低未命中缓存导致的长延迟技术的研究

杨洁[1)]

[1)](华中科技大学，武汉光电国家研究中心，湖北省武汉市 435400)

摘 要 深度缓存层次结构和相对较慢的主存导致的高加载延迟是单线程性能的重要限制因素，如何降低访存延迟一直是计算机体系结构领域的重点研究方向之一。迄今为止，计算机架构师在此领域取得了累累硕果，几乎每个高性能处理器都支持多种手段来降低处理器与内存速度不匹配所带来的影响，这些应对方法大致可以分为两种类型，即延迟避免技术和延迟容忍技术。前者主要通过捕获不同应用程序的各种访问模式，对后续所需的数据进行推测和预取，从而有效地隐藏了高的存储器访问延迟；后者则在顺序指令流中的延迟访存时，继续执行独立指令，从而避免流水线停顿。这些方法都已特定的方式提升了系统的效率。在本次调查中，主要对两种不同技术的最新研究成果进行了介绍，并对不同技术的优缺点进行了评估，阐明了不同技术的局限性。

关键词 访存延迟；数据预取；乱序执行

The Survey of Reducing High Load Latency Caused by Missing Cache

Jie Yang[1)]

[1)](Wuhan National Laboratory for Optoelectronic, Huazhong University of Science and Technology, Wuhan, Hubei)

Abstract High loading latency caused by deep cache hierarchy and relatively slow main memory is an important limiting factor for single-thread performance. How to reduce memory access latency has always been one of the key research directions in the field of computer architecture. So far, computer architects have made great achievements in this field. Almost every high-performance processor supports multiple methods to reduce the impact of the mismatch between processor and memory speed. These solutions can be roughly divided into two types, namely delay-avoiding techniques and delay-tolerant techniques. The former mainly captures various access patterns of different applications, speculates and prefetches the subsequent required data, thus effectively hiding the high memory access latency; Execution of independent instructions continues, avoiding pipeline stalls. These methods have improved the efficiency of the system in a specific way. In this survey, the latest research results of two different technologies are mainly introduced, and the advantages and disadvantages of different technologies are evaluated, and the limitations of different technologies are clarified.

Key words load latency, data prefetching, out of order execution

1 引言

诸如媒体流和网络搜索的服务器，需要同时向数百万用户提供服务，是一类非常重要的应用程序。这类服务器往往运行在在由处理器支持的大规模数据中心上，其中的处理器基本上会针对低延迟和服务质量保证进行调整，通常包括高的时钟频率、积极的预测策略和深度流水线等等，以尽可能快地运行服务器应用程序，满足最终用户的延迟要求

然而，内存系统中限制了服务器处理器的性能发挥，成为了应用程序的性能瓶颈。由于服务器需要对大量数据进行操作，它们会因此产生活动内存工作集。而服务器处理器中，容量有限的片上缓存对此相形见绌，因此活动内存工作集往往需要驻留在片外内存中，导致应用程序经常会错过片上缓存中的数据，未命中缓存层次结构的内存加载会受到较长的 DRAM 延迟的影响，从而导致流水线的停滞，对每周期的指令数(IPC)产生不利影响。

迄今为止，已经有许多研究人员针对上述问题展开了深入研究，并提出了两种主要方法来应对这一挑战：延迟避免(latency-avoiding)和延迟容忍(Latency-tolerating)。延迟避免

技术利用规则或不规则模式下的预取以及辅助线程，在数据需求负载发生之前将数据预测性地读入缓存，有效地隐藏了高内存访问延迟。延迟容忍技术，如乱序执行(Out-of-order execution)，会在顺序指令流出现高延迟负载后，继续执行独立的指令，避免管道阻塞。考虑指令关键程度的系统进一步扩展了重新排序指令的机会，并增加了在出现延迟负载的情况下可以重新排序的指令数量。

如今，几乎所有的高性能处理器都会采用延迟避免和延迟容忍技术，来尽可能降低由处理器与内存系统之间的性能差距而导致的处理器空闲等待时间，提高处理器的利用率。在研究文献中，同样有无数关于这两种技术的建议，每个建议都以一种特定的方式来提高缓存的命中率，或是降低缓存未命中给应用程序所带来的负面影响。

在这项调查中，我将从延迟避免和延迟容忍两个方面来对目前有关降低访问 DRAM 而带来的长延迟对于计算机运行速度的影响。本文的其余部分组织如下：第二章简要介绍了每一类延迟避免技术的背景，并介绍和分析了不同延迟避免技术的最新研究；第三章描述了延迟容忍技术的发展，并着重介绍了基于关键性的预取技术；第四章为结论，对全文进行一个分析和总结。

2 延迟避免

在本节中，我将对现在的延迟避免技术进行整体性的介绍，并详细介绍现有的最先进的延迟避免技术以及其机制。常见的延迟避免技术包括以下 5 类。

2.1 跨距预取器(Stride Prefetcher)

跨距预取器广泛用于商业处理器(例如 IBM Power4、Intel Core、AMD Opteron、Sun UltraSPARC III)，并且已被证明在桌面和工程应用中非常有效。跨距预取器检测呈现跨距访问模式的流(即连续地址序列)，并通过将检测到的跨距与最后观察到的地址相加来生成预取请求。虽然早期的跨度预取器只捕获由恒定跨度分隔的访问模式，但随着时间的发展，目前跨距预取器已经可以捕获多个或可变跨度的访问模式。

偏移预取器是跨步预取器的一种演变，这种预取器不会尝试检测跨步流。相反，每当处理器请求缓存块时，偏移预取器将预取由 k 个缓存行隔开的缓存块，其中 k 是预取偏移。换句话说，偏移预取器不再将所访问的地址与任何特定流相关联，而是单独处理地址，并根据预取偏移量，对每个访问的地址发出预取请求。值得注意的是，偏移预取器可以基于应用的行为动态地调整预取偏移。

Pugsley 等率先提出了一种偏移量预取器 SBP(Sandbox Prefetcer)[1]，它尝试动态查找产生准确预取请求的偏移量。为了找到这样的偏移量，SBP 定义了评估周期，在该评估周期中，它评估了-n 到+n 的多个预定义偏移量的预取精度。对于每个预取偏移量，SBP 为其关联了一个分数值，当评估周期结束时，只有分数值超过某个阈值的偏移量才被认为是准确的偏移量。然而，Pugsley 没有考虑预取的及时性。只有在预取隐藏了大量未命中延迟的情况下，发出准确的预取才有帮助。

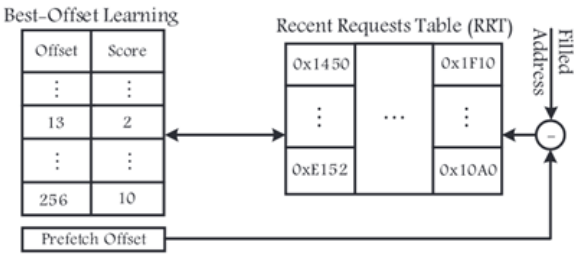

图 1 最佳偏移预取器（BOP）的硬件结构

在 Pugsley 的基础上，Pierre Michaud 提出了一种改进的偏移量预取器 BOP(Best-offset hardware prefetching)[2]，BOP 调整了 SBP 并尝试选择产生及时预取请求的偏移量。图 1 显示了 BOP 的硬件结构。与 SBP 类似，BOP 评估各种偏移量的效率，并选择最佳偏移量进行预取。但与 SBP 不同的是，BOP 促进了能够及时生成预取候选者的偏移量，而不仅仅是准确的候选者。BOP 的基本思想是：“如果 k 是 A 行的及时预取偏移量，则 A-k 行应该是最近被访问的。”也就是说，在其预取候选中，被应用程序使用的时间不比它们生成的时间长得多的偏移量会被认为是合适的偏移量，并且相应地，它们的分值被递增。为了评估使用各种偏移量发出的预取请求的及时性，BOP 将 SBP 中的 Bloom Filter 替换为集关联最近请求表(RRT)。RRT 的大小被故意选择为较小，以仅保留最近的请求。对于每个触发事件 A，在偏移量 k 的评估周期内，如果 A-k 行命中了 RTT，则偏移量的分数递增。换言之，在偏移量 k 的评估周期下，如果第 A 行被处理器请求，并且 A-k 行命中了 RTT，则其被解释为“对于偏移量 k，如果已经对 A-k 行上发出预取请求，则其预取请求(A-k+k=A)将是及时的。”因此，k 被标识为其预取请求与处理器的需求匹配的偏移量，进而其分数递增。

与 SBP 评估 n 在-n 到+n 范围内的所有偏移量不同，BOP 仅仅评估 46 个（几乎随机）根据经验选择的常数偏移量。BOP 的其他部件和功能则与 SBP 相似。

2.2 空间预取器(Spatial Prefetcher)

空间预取器通过依赖空间地址相关性（即多个存储器区域之间的访问模式的相似性）来预测未来的存储器访问。由于应用程序使用具有规则和固定布局的数据对象，并且当数据结构被遍历时，访问将多次发生。空间数据预取器将存储器地址空间划分为固定大小的部分，称为空间区域，并学习

这些区域上的存储器访问模式。然后，当应用程序访问相同或相似的空间区域时，学习的访问模式用于预取未来的内存引用。

空间数据预取器具有较低的区域开销，因为它不使用完整的地址，而是以存储偏移量（即，块地址与空间区域开始的距离）或增量（即，落入空间区域的两个连续访问的距离）作为元数据信息。空间数据预取器的另一个同样显著的优势是消除强制缓存未命中的能力。强制缓存未命中是应用程序性能下降的主要原因，例如，扫描的工作负载，其中扫描大量数据会产生大量无法被缓存捕获的不可见内存访问。通过利用在过去的空间区域中观察到的模式到新的未观察到的空间区域，空间预取器可以减轻强制缓存未命中，从而显著提高系统性能。

空间数据预取的关键限制是它不能预测指针追逐导致的缓存未命中。由于动态对象可能被分配到存储器中的任何地方，所以指针追逐访问不一定表现出空间相关性，从而产生大量相关的高速缓存未命中，而空间预取器对这些未命中几乎无能为力。

Stephen Somogyi 等人将元数据访问足迹与触发访问的“PC+偏移量”相关联，提出了一种强大的空间预取器 SMS(Spatial Memory Streaming)[3]。每当应用程序第一次请求空间区域时，SMS 就开始观察和记录对该空间区域的访问。每当空间区域不再被使用时(即，空间区域的相应块开始从高速缓存中被逐出)，SMS 将观察到的访问的信息存储在其名为模式历史表(PHT)的元数据表中。只要将来再次发生事件，就使用 PHT 中的访问模式。由于 SMS 选择触发访问的“PC+偏移量”作为与访问模式相关的事件。这样一来每当“PC+偏移量”再次出现时，相关的访问模式历史就被用于发出预取请求。

Mohammad Bakhshalipour 等人在 SMS 的基础上，提出了使用一个统一的元数据表记录多个触发访问的事件(PC+地址和 PC+偏移量)的空间数据预取器 Bingo[4]。类似于 SMS，当处理器访问空间区域时，Bingo 也会使用一个小的辅助存储来记录空间模式。Bingo 在其辅助存储中为页面分配一个条目并为其记录足迹。当一个条目在页面的驻留结束时(即，每当来自页的块被无效或从高速缓存逐出时)，Bingo 将所记录的模式传送到其历史表，并释放辅助存储器中的对应条目。

但与 SMS 不同的是， Bingo 使用“PC+地址”和“PC+偏移量” 作为触发访问来进行预取。逻辑上，Bingo 维护两个不同的历史表来，一个历史表维护每个“PC+地址”之后观察到的访问足迹历史，而另一个表则维护与“PC+偏移量”相关的足迹元数据。但为了有效地消除两个历史表在存储中的冗余，基于短事件会承载在长事件中的思想，Bingo 实际上仅维护一个历史表。也就是说，“PC+偏移量”的信息包含在“PC+地址”中，因此通过 “PC+地址”的信息就可以得到“PC+偏移量”的信息。为了利用这种现象，Bingo 只存储长事件的历史，但同时使用长事件和短事件进行查找，其具体结构如图 2 所示。对于 Bingo 来说，历史表存储了在每个

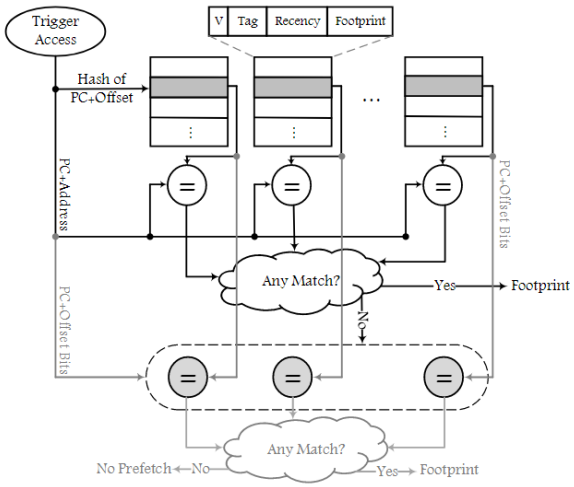

图 2 空间数据预取器 Bingo 的硬件结构

“PC+地址”事件之后观察到的足迹，但是使用触发器访问的“PC+地址”和“PC+偏移量”来查找，以便在不丢失预取机会的同时提供高准确性。

Bingo 首先使用触发访问的“PC+地址”查找表。如果找到匹配，则将使用相应的占用空间元数据发出预取请求。否则，Bingo 会使用触发访问的“PC+偏移量”来继续查找该表。由于'PC+地址'和'PC+偏移量'都映射到同一个集合，所以不需要检查一个新的集合。在这种情况下，并不是条目中存储的标签的所有位都需要匹配，而是只有“PC+偏移量”位需要匹配。通过这种方式，作者将每个足迹与多个事件关联起来(即'PC+地址'和'PC+偏移量')，但将足迹元数据仅与其中一个(较长的那个)存储在表中，以减少存储需求。因为元数据占用只存储一次，所以 Bingo 会自动消除冗余。

Rahul Bera 等人提出了一种与不同于上述预取器的轻量级的双空间位模式预取器 DSPatch[5]。DSPatch 在每个存储区域学习两个位模式，并将它们与基于程序计数器(PC)的签名相关联。一种比特模式(称为 CovP)倾向于更高的覆盖范围，它计算最近观察到的空间程序对物理页的访问位模式的或运算。或运算将比特添加到已有的比特模式中，并增长比特模式以获得更高的覆盖率，直到达到某一阈值。另一种位模式(称为 ACCP)偏向于更高的精度。它计算覆盖偏置比特模式和当前观察到的对物理页面的程序访问比特模式的和运算。和运算减少了设置位以最大限度地提高精度，但由于 AccP 派生自 CovP，因此覆盖范围保持在可控范围内。

如图 3 所示，DSPatch 主要由两个硬件结构组成：页面缓存

(PB)和签名模式表(SPT)。PB 的目的是在程序访问物理页面时记录观察到的空间位模式。SPT 的目的是通过将来自先前观察到的两个调制空间位模式(CovP 和 ACCP)与触发器 PC 关联到页面中来存储它们。因此，DSPatch 观察每个

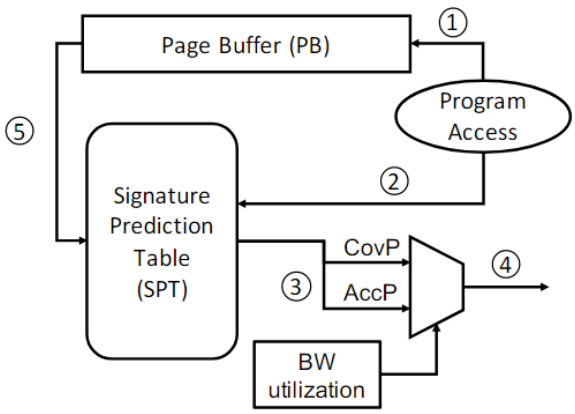

图 3 双空间位模式预取器 DSPatch 的硬件结构

物理页面的访问程序，通过将空间位模式与触发 PC 签名相关联来以与页面无关的方式学习整体程序访问模式。

当新的物理页面被访问时，SPT 与触发 PC 进行查找，以检索两个调制位模式：CovP 和 AccP。DSPatch 的关键目标是根据 DRAM 带宽利用率动态调整预取，以实现更高的覆盖范围或更高的精度。使用从内存控制器广播到所有核的简单的 2 位带宽利用率信号，DSPatch 选择 CovP(当内存带宽利用率低时)或 ACCP(当内存带宽利用率高时)位模式来驱动预取。

DSPatch 在内存控制器上使用一个简单的计数器来跟踪内存带宽利用率，该计数器对在(4×TRC)个周期的时间窗口内发出的 DRAM 列访问(CAS)命令的数量进行计数(其中 TRC 是两个 DRAM 行激活之间允许的最小时间)。为了在跟踪中包括滞后，在每个窗口之后都设置了计数器。通道的数量和每个通道的宽度决定了峰值 DRAM 带宽，以及每个 TRC 窗口中可能的 CAS 命令的峰值数量。将该计数器存储到峰值带宽的四分位数(25%、50%和 75%)，并在每个 TRC 周期中，将计数器的值与 3 个四分位数阈值中的每一个进行比较，从而产生表示当前带宽利用率落入哪个四分位数的 2 比特量化值(例如，3 表示高于 75%的带宽利用率，而 0 表示低于 25%的带宽利用率)。这个 2 比特量化的带宽利用率值被广播到所有核心，并在 DSP 匹配算法中用作当前内存带宽利用率的代表。

2.3 时间预取器(Temporal prefetcher)

时时间预取指的是通过重放过去的缓存未命中序列，以避免未来的缓存未命中。时间数据预取器按照数据未命中的出现顺序记录数据未命中序列，并使用所记录的历史序列来预测未来的数据未命中。在新的数据缺失时，它们搜索历史序列并找到匹配条目，并在匹配之后重放数据缺失的序列，以尝试消除潜在的未来数据缺失。在 IBM Blue Gene/Q 中实现了时间预取的调优版本，称为列表预取

时间预取是消除相关高速缓存未命中的长链的理想选

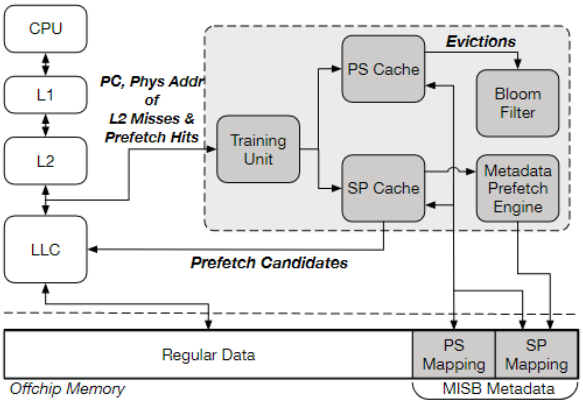

图 4 时间数据预取器 MISB 的硬件结构

择，这在指针追逐应用程序(例如，OLTP 和 Web)中很常见。依赖高速缓存未命中是一种存储器操作，它会导致高速缓存未命中，并且依赖于先前高速缓存未命中的数据。因为其导致的两个未命中都是顺序获取的，这种未命中对应用程序的执行性能有显著影响，并阻碍处理器向前推进，。由于相关未命中之间缺乏跨距/空间相关性，因此跨距数据预取器和空间数据预取器通常不能预取这种未命中。时间预取器通过记录和重放数据未命中序列，则可以预取相关高速缓存未命中，从而带来显著的性能改进。

限制时间预取技术使用的关键在于其表现出的低精度，因为时间预取器不知道流在哪里结束。也就是说，在时间预取的基础上，没有关于何时应该停止预取的信息。因此，时间预取器不断发出许多预取请求，直到另一个触发事件发生，导致大量的过度预测。此外，由于临时预取器依赖地址重复，它们无法防止强制未命中(未观察到的未命中)的发生，只能预取在过去至少观察到一次的高速缓存未命中。然而，存在许多重要的应用程序(例如，DSS)，其大多数高速缓存未命中在应用程序的执行期间仅发生一次，时间预取器无法为这种应用程序带来任何增益。最后，时间预取器需要相当多的存储空间来存储地址关联。为了减少元数据需求，一些时间预取器放弃地址关联，转而学习较弱形式的关联，例如增量关联、标记关联或上下文-地址对关联，但这些简化限制了可以学习的内存访问模式的范围。时间还有一些预取器通过在片外存储器中存储元数据来利用地址相关性，但片外元数据的使用限制了此类时间预取器的商业可行性。

针对元数据存储开销过大这一问题，Hao Wu 等人在不规则流缓冲区 ISB(Irregular Stream Buffer)的基础上，提出了一种使用简单的元数据预取器来提供元数据缓存的时间数据预取器 MISB(Managed ISB)[6]，其结构如图 4。MISB 的

元数据管理方案有三个组件，包括元数据缓存、元数据预取器和避免发出虚假元数据请求的元数据过滤器，这三个组件共同使 MISB 能够有效地管理元数据。

元数据缓存非常重要，因为片上元数据访问具有最小的延迟，并且不会引起片外流量。因此，与 ISB 一样，MISB 有两个片上缓存，用于存储其总的片外元数据的一小部分。其中，PS 缓存存储从物理地址到结构地址的映射，而 SP 缓存存储结构到物理地址的映射。虽然缓存处理许多元数据请求的延迟和流量问题，但缓存本身无法隐藏所有元数据访问的延迟，因此为了进一步隐藏延迟，MISB 使用了元数据预取器。这种预取器背后的关键见解是，由于时间流在结构化地址空间中按顺序布局，因此可以通过在 SP 缓存上部署简单的下一行预取器来准确预取元数据。最后，由于 MISB 的 PS 缓存中的未命中会产生大量的伪元数据流量，这样的请求增加了流量，但没有提供任何好处。MISB 使用 Bloom Filter 来过滤这些请求。

在 MISB 的基础上，Hao Wu 等人重新调整芯片上缓存空间的用途，进一步提出了一种时间数据预取器 Triage[7]。为了有效地利用宝贵的片上缓存空间，首先，Triage 学习 PC 本地化的相关地址对，并将它们记录在表中。虽然表是组织芯片外元数据的糟糕选择，但它们的空间效率使它们成为组织芯片内元数据的理想选择。此外，Triage 还使用了压缩地址表示来进一步减少元数据占用。

其次，作者通过观察得到了三个规律。第一，大多数元数据重用可归因于少数元数据条目。第二，即使在频繁重复使用的元数据条目中，也很少有预取是非冗余的。第三，元数据应该以精细的粒度进行管理和驱逐。基于以上结果，作者修改并使用了一种最先进的缓存替换策略 Hawkey，它从过去内存引用的最优解决方案中学习。在 Triage 中，只有当元数据产生缓存中未命中的预取时，才对 Hawkey 策略进行积极训练。

最后，Triage 还调整了元数据存储区的大小。为了避免应用程序数据和元数据之间的冲突，Triage 通过为数据和元数据分配不同的路径来对末级缓存进行分区。通过这三种修改，与使用 2MB 缓存且没有预取的基线相比，分流使流量增加了 59.3%，而 STMS、Domino 和 MISB 分别增加了 482.9%、482.7%和 156.4%的流量。

2.4 超前执行(Runahead Execution)

超前执行通过准确地预取长延迟加载来提高处理器性能。当长等待加载导致指令窗口填满并停止流水线时，处理器触发超前执行。处理器不再仅仅等待，而是移除阻塞的长等待加载，并推测后续指令，以发现未来独立的长延迟加载并显示内存级并行性(MLP)。当停止加载返回时，处理器终止超前执行并恢复正常操作。由于超前执行通过提前查看应用程序的代码来生成内存负载，因此它生成的预取请求是准确的，从而带来显著的性能优势。

但是，超前执行的性能优势受到其预取覆盖率和与推测代码执行相关的开销的限制。一方面，预取覆盖范围与在提前运行模式下生成的有用预取请求的数量有关，超前运行模式下的预取覆盖率越高，超前执行的性能优势就越大。另一方面，推测性代码执行增加了保存和恢复状态的开销，并在超前执行后将流水线回滚到适当的状态以恢复正常操作。这些开销的性能损失越小，收益就越高。因此，要最大化提前运行执行的性能优势，需要(1)最大化每个提前运行间隔的有用预取次数，以及(2)限制提前运行模式和正常执行之间的切换开销。

为了缓解超前执行的局限性，Ajeya Naithani 等人观察并分析了先前的超前执行方案，并提出了精确提前执行 PRE(Precise Runahead Execution)[8]。PRE 通过在超前运行模式下预取所有停滞切片(与超前运行缓冲区不同)并只执行导致加载的指令链(与原始超前建议不同)，从而提高了预取覆盖率。此外，PRE 在进入提前运行模式时不会释放处理器状态，因此在恢复正常模式时不需要刷新和重新填充流水线，降低了调用提前执行的成本。

与在现有技术中一样，PRE 在全窗口停顿时被调用。在通过寄存器别名表 full-ROB 对指令的 PC 进行检查点之后，PRE 进入超前运行模式。在超前运行执行期间，当指令从解码单元到达时，PRE 动态地识别作为潜在停滞切片一部分的指令，并且内核推测性地执行它们。为了识别潜在停滞切片，PRE 跟踪在停滞切片表(SST)的新高速缓存中形成停滞切片的各个指令。PRE 在解码阶段之后访问 SST。SST 是只包含指令地址(即 PC)的全关联高速缓存。如果指令地址命中 SST，则该指令是停滞切片的一部分。使用 SST 预过滤并推测性地执行跟随停滞窗口的所有停滞切片。在指令解码之后，PRE 只执行命中 SST 的指令，因为它们是生成未来加载所必需的。PRE 实现了与提前运行缓冲区一样的过滤提前运行执行的好处，因为它只执行停滞切片。由于 SST 存储所有停滞切片，所以 PRE 设法执行所有可能的停滞切片，这导致了大大改进的预取覆盖率。

2.5 缓存级别预测(Cache Level Prediction)

计算机缓存层次结构的级别通常是按顺序查找的，从第一级高速缓存(L1)开始，经过第二级和第三级高速缓存。如果在任何缓存中都没有找到该数据，则从内存中获取该数据。深度缓存层次结构通常会提高性能，但当缓存不能成功过滤请求时，可能会导致更高的加载延迟，只会增加查找延迟。 上文所述的预取器通过在执行访存指令之前在不同的缓存级别之间移动数据，在一定程度上减轻了逐级查找的延迟影响，但许多加载仍然面临顺序查找延迟；而并行查找缓

存会增加能源消耗，并需要过度配置标记阵列端口和可能的片上带宽或完全重新设计整个层次结构。

缓存级别预测器提供了两个方面的最佳结果：减少了非顺序(并行)查找的访问延迟，同时保持了顺序层次结构的低

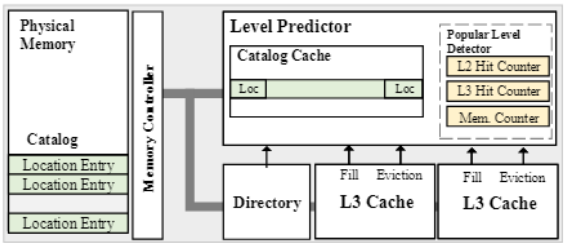

图 5 级别预测器的体系结构

访问成本和简单性。通过这种方式，级别预测(LP)提高了那些在有高级预取器和替换策略的情况下，缓存中不可避免地未命时的访存性能。

Majid Jalili 等人设计了一个有效且低成本的级别预测器[9]，该预测器在 L1 高速缓存未命中路径上的每个核中运行，并且在结合预取器时显著优于高速缓存未命中预测器。如图 5 所示，它由两个组件组成：(1)全局位置图(LocMap)的元数据缓存，它保存关于每个内存块的级别的可能过时的信息；(2)一个紧凑的基于历史的流行级别检测器预测器，用于元数据缓存未命中。LocMap 需要每 512 位缓存块 2 位来指示其内存级别。它在系统保留的物理内存中被组织为平面表。它的访问粒度与数据缓存相同-在元数据缓存未命中后，每次访问 LocMap 时都会获取 256 个缓存块的信息。历史预测器是 3 个 32 位的寄存器，用来计算每个级别的命中次数，在每一次 L1 未命中时进行查询。

作者使用 3 个计数器，每个高速缓存级别和主内存一个。当命中某一级别时，相应的计数器递增 1，其他计数器递减 1。当需要预测时，计数器被排序，最上面的被选为第一个候选者。如果其计数器高于阈值，则只选择该级别作为要查找的级别。否则，计数器第二高的级别也被视为可能的目的。若前两个级别的计数值仍然低于阈值，则继续访问计数器第三高的级别。

通过这种缓存级预测器，将不必要的访问过滤到中间级别，从而减少了累积的未命中延迟。与基线相比，此系统提高了 7.8%的 IPC。

3 延迟容忍

在本节中，我将对现在的延迟容忍技术进行整体性的介绍，并详细介绍现有的最先进的延迟容忍技术以及其机制。常见的延迟容忍技术包括以下 2 类。

3.1 乱序执行(Out of order)

乱序执行是一种应用在高性能微处理器中来利用指令周期以避免特定类型的延迟消耗的范式。在这种范式中，处理器根据输入数据的可用性确定执行指令的顺序，而不是根据程序的原始数据决定。在这种方式下，可以避免因为获取下一条程序指令所引起的处理器等待，取而代之的处理下一条可以立即执行的指令。

实现乱序执行的方法主要有两种，软件方式和硬件方式。软件的方式通过编译器与体系结构的强耦合，在编译阶段就生成好无相互依赖，易于处理器调度的指令。在编译阶段进行指令重排又被称为静态指令调度，优点是软件实现可以更灵活，通常软件也可以有足够的存储空间来分析整个程序，因此可以获得更优的指令排布。当然缺点也是显而易见的，由于编译器需要深入地了解体系结构相关的信息，如指令延迟和分支预测惩罚等，对可移植性造成了很大的困难。因此现代处理器更加常用的是硬件方式。

硬件方式主要是通过寄存器重命名来消除读—读和写—写假依赖。寄存器重命名就是对不同指令调用的相同寄存器使用不同的物理硬件存储，在写回阶段再对这些指令和寄存器进行排序，这样这些假依赖就不再是产生流水线气泡的原因。硬件方式的优点在于降低了编译器的体系结构耦合度，提高了软件编写的便捷性，通常硬件乱序执行的效果也不比软件的差。而缺点在于依赖分析和寄存器重命名都需要耗费宝贵的片上空间和电力，但对于性能的提升却没有相应的大。

3.2 基于关键性的预取器(Criticality-based prefetcher)

考虑指令关键程度的系统进一步扩展了重新排序指令的机会，并增加了可在延迟加载的情况下重新排序的指令数量。这里所谓的关键性，就是造成 LLC cache 未命中的程度，访存切片(Load Slice)的地址生成指令的数量和这些访存切片的关键路径长度。在整个内存层次结构中，利用 MLP(memory level parallelism)，通过重叠内存访问来将后一个请求的延迟隐藏在前一个请求的“阴影”中，对于性能至关重要。为了最大限度地降低 MLP 开发的能源成本，一种名为 sOoO(slice out of order)核的新型核构建在高能效有序执行的基础上，并为 MLP 提取添加了刚刚好的 OoO 支持。这些核心首先构造包含导致加载和/或存储的地址生成指令的 MLP 生成指令切片。切片相对于指令的其余部分被无序执行，切片和其余指令本身仍按顺序执行。Rakesh Kumar 等在 SOoO 的基础上，提出了 Freeway[10]。这是一种基于新的依赖感知片执行策略的 SOoO 方法，它的非关键指令由有序流水线执行，并且允许加载旁路。

为了增加 MLP，Freeway 放弃了 FIFO(first in first out)切片执行，并允许独立切片以最少的额外硬件相对于依赖切片无序执行。为了实现切片与切片之间的无序执行，Freeway

跟踪硬件中的片依赖关系，并将依赖片与独立片分开。Freeway 使用片依赖信息通过为独立片独占预留 B-IQ(bypass queue)来加速独立片的执行。为了处理依赖片，利用第三节中的见解，Freeway 引入了一个新的 FIFO 指令队列，称为 Y-IQ(yield queue)。然后，依赖切片可以在 Y-IQ 中等待，导致独立切片执行，直到它们准备好执行。依赖切片大多按照程序顺序准备好，因此，准备好的依赖切片应该很少落后于未准备好的片。这一特性允许 Freeway 使用简单的

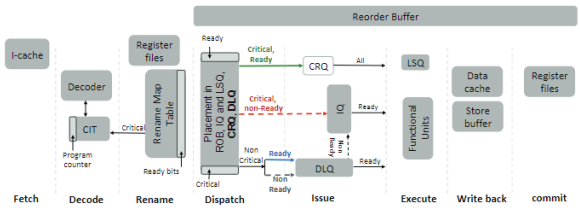

图 6　基于关键性的预取器 DNB 的体系结构

硬件通过执行来自 B-IQ 和 Y-IQ 的大部分切片来提升 MLP，而不会出现任何停顿。Freeway 的性能比最先进的 sOoO 核心高出 12%，并且在完全 OoO 执行的 MLP 极限的 7%之内。

Mehdi Alipour 等人则提出了一种利用指令临界性就绪性来改善指令调度的方法 DNB[11]，它将指令队列划分为包含就绪指令、非就绪指令、关键指令和非关键指令的更小的子队列，以提高调度能量效率，降低了指令队列(IQ)宽度和深度。

对于关键指令和就绪指令，DNB 绕过 IQ 以降低 IQ 容量和端口压力以及 IQ 读/写；通过将非关键指令和非就绪指令放置在 FIFO 中来延迟它们，从而降低 IQ 容量压力；对于到达延迟 FIFO 队列头部时准备就绪的延迟指令，DNB 同样绕过 IQ 来降低 IQ 端口压力，以及 IQ 读/写。在 DNB 中，只有关键和非就绪指令被直接放置在昂贵的 Content-Addressable Memory (CAM)中，因为这些指令是一旦准备好就必须执行的指令。

图 6 概述了 DNB 体系结构：DNB 增加了用于延迟指令的 FIFO 队列(Delay-Queue，DLQ)、用于将关键和就绪指令的后端执行与前端提取分离的 FIFO 队列(Critical-Ready-Queue，CRQ)，以及用于迭代向后相关性分析(IBDA)以确定指令关键程度的关键指令表(Critical Instruction Table，CIT)。LTP 和 DNB 共有的组件显示为浅灰色，添加部分显示为白色。

然而，上述硬件实现的关键性预取器都面临两个问题，一是指令调度困难(NP 难问题)，而是元数据存储开销大且性能一般。相比之下，软件技术支持基于每个单独加载的执行频率、未命中率和内存级并行性(MLP)属性将延迟加载分类为关键指令的复杂策略。此外，可以考虑被称为访存切片(Load Slice)的地址生成指令的数量和这些访存切片的关键路径长度来确定临界性。与只能处理有限大小的访存切片的硬件技术相比，软件技术不受元数据存储开销的限制。最后，软件技术通过内存来观察相关性，这是实现精确和全面的加载片的关键能力。

Heiner Litz 等提出了一种软件实现的关键性预取器 CRISP[12]，作者通过软件实现解决了这两个问题，唯一的硬件需求是一种标记指令并以此为优先级调度指令的机制。由于一条 Load/Store 指令往往与其他指令存在依赖关系(如 WAW，RAW)，存在依赖关系的这一系列指令可以被看作一条访存切片，CRISP 的主要工作便是查找并重新调度关键性访存切片。

CRISP 首先运行目标程序，通过监控系统(如 GWP 或作者之前提出的 AsmDB)获取运行情况，分析数据流并标记关键访存切片，将访存切片通过软件预取注入(software prefetch injection)的方式内嵌到汇编语言。然后，CRISP 给所有指令增加一个新位域 priority 用于标识其是否是一条关键指令，初始化为 0。所有内嵌的预取指令 priority 位域设为 1，在 ROB 中的 RS(reservation station，用于调度并将指令发射至功能单元)综合考虑指令关键性及先后关系调度。具有关键性的、先来的指令优先发射。

CRISP 不仅关注访存关键指令，还关注分支关键指令，因为很多数据集因为错误的分支预测而遭受性能瓶颈。简单来讲，CRISP 根据造成流水线停顿的程度决定是否预取并优先调度该条指令。

4 总结

在过去四十年中，降低访存开销一直是一个很有吸引力的领域，大量的研究表明了其重要性与有效性，降低访存延迟的技术已经成为了现代高性能处理器中不可分割的一部分。

在本文中，主要介绍和评估了几种不同的降低访存延迟的方法，这些方法大体上可以分为延迟避免和延迟容忍这两种不同的解决方案，并且对不同方法的最新研究进行了介绍，分析了不同方法的创新点和局限性。同时，即使是最先进的方法，与理想的方案比仍有很大差距，因此这一领域仍然需要进一步的研究。未来工作的一个方向是弥合表现最好的预取器的性能和理想方案之间的差距。

参 考 文 献

[1] S. H. Pugsley et al. "Sandbox Prefetching: Safe run-time evaluation of aggressive prefetchers." 2014 IEEE 20th International Symposium on

High Performance Computer Architecture (HPCA), Orlando, FL, USA, 2014, pp. 626-637, doi: 10.1109/HPCA.2014.6835971.

[2] P. Michaud. "Best-offset hardware prefetching." 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA), Barcelona, Spain, 2016, pp. 469-480, doi: 10.1109/HPCA.2016.7446087.

[3] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi and A. Moshovos, "Spatial Memory Streaming," 33rd International Symposium on Computer Architecture (ISCA'06), Boston, MA, USA, 2006, pp. 252-263, doi: 10.1109/ISCA.2006.38.

[4] M. Bakhshalipour, M. Shakerinava, P. Lotfi-Kamran and H. Sarbazi-Azad, "Bingo Spatial Data Prefetcher," 2019 IEEE International Symposium on High Performance Computer Architecture (HPCA), Washington, DC, USA, 2019, pp. 399-411, doi: 10.1109/HPCA.2019.00053.

[5] Rahul Bera, Anant V. Nori, Onur Mutlu, and Sreenivas Subramoney. DSPatch: Dual Spatial Pattern Prefetcher. In Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '52). Association for Computing Machinery, New York, NY, USA, 2019. 531–544. https://doi.org/10.1145/3352460.3358325

[6] H. Wu, K. Nathella, D. Sunwoo, A. Jain and C. Lin, "Efficient Metadata Management for Irregular Data Prefetching," 2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA), Phoenix, AZ, USA, 2019, pp. 1-13.

[7] Hao Wu, Krishnendra Nathella, Joseph Pusdesris, Dam Sunwoo, Akanksha Jain, and Calvin Lin. 2019. Temporal Prefetching Without the Off-Chip Metadata. In Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '52). Association for Computing Machinery, New York, NY, USA, 996–1008. https://doi.org/10.1145/3352460.3358

[8] A. Naithani, J. Feliu, A. Adileh and L. Eeckhout, "Precise Runahead Execution," 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA), San Diego, CA, USA, 2020, pp. 397-410, doi: 10.1109/HPCA47549.2020.00040.

[9] M. Jalili and M. Erez, "Reducing Load Latency with Cache Level Prediction," 2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA), Seoul, Korea, Republic of, 2022, pp. 648-661, doi: 10.1109/HPCA53966.2022.00054.

[10] R. Kumar, M. Alipour and D. Black-Schaffer, "Freeway: Maximizing MLP for Slice-Out-of-Order Execution," 2019 IEEE International Symposium on High Performance Computer Architecture (HPCA), Washington, DC, USA, 2019, pp. 558-569, doi: 10.1109/HPCA.2019.00009.

[11] M. Alipour, S. Kaxiras, D. Black-Schaffer and R. Kumar, "Delay and Bypass: Ready and Criticality Aware Instruction Scheduling in Out-of-Order Processors," 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA), San Diego, CA, USA, 2020, pp. 424-434, doi: 10.1109/HPCA47549.2020.00042.

[12] Heiner Litz, Grant Ayers, and Parthasarathy Ranganathan. 2022. CRISP: critical slice prefetching. In Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22). Association for Computing Machinery, New York, NY, USA, 300–313. https://doi.org/10.1145/3503222.3507745