基于指令流混合模式学习的缓存预取算法

王玉庆[1,2],杨秋松[1],李明树[1]

(1. 中国科学院软件研究所基础软件国家工程研究中心,北京 100190; 2. 中国科学院大学,北京 100049)

摘　要： 近期缓存预取算法的研究热点是使用基于模式识别的预测技术,例如Lookahead,推算访存请求的地址. 此类算法一方面很难学习访存行为中的依赖缓存失效,另一方面不能精确控制预取请求发送和写回的时机. 为了解决上述问题,本文提出了一种基于分支预测技术和混合模式学习的缓存预取(Instruction Flow Based Hybrid Prediction,IFBHP)算法. 使用分支预测技术识别程序未来指令流中的访存指令流,通过多种地址关联模式的学习逐一计算访存指令流中每条指令的地址,写入访存地址队列. 使用阈值评估未来指令流进入处理器主流水线的时刻,精确控制指令流所对应的预取请求的发送和写回. 实验表明,本文算法相比STeMS(Spatio-Temporal Memory Streaming)算法、ISB++(Irregular Stream Buffer++)算法、SANGAM算法、IPCP(Instruction Pointer Classifier based spatial Prefetching)算法一级数据的读操作缓存失效次数分别平均减少31.58%,28.85%,17.85%,11.48%;本文算法相比STeMS算法、ISB++算法、SANGAM算法、IPCP算法一级数据的写操作缓存失效次数分别平均减少31.58%,28.85%,17.85%,11.48%.

关键词： 缓存预取;分支预测;时间关联模式;步长模式;指令流

基金项目： "核高基"国家科技重大专项(No.2014ZX01029101-002);中国科学院战略性先导科技专项(No.XDA-Y01-01)

中图分类号： TP302　　**文献标识码：** A　　**文章编号：** 0372-2112(2023)02-0342-13

电子学报URL:http://www.ejournal.org.cn　　**DOI:**10.12263/DZXB.20210273

A Cache Prefetching Mechanism Based on Hybrid Pattern Learning of Instruction Flow

WANG Yu-qing[1,2], YANG Qiu-song[1], LI Ming-shu[1]

(1. *National Engineering Research Center for Fundamental Software*, *Institute of Software*, *Chinese Academy of Sciences*, *Beijing* 100190, *China*;
2. *University of Chinese Academy of Sciences*, *Beijing* 100049, *China*)

Abstract: Pattern recognition mechnisms,such as Lookahead, have been used to calculate the addresses of prefetching in cache prefetchers. However, these mechanisms cannot dispose cache dependent missing. At the same time, they cannot send and write back prefetching requests to cache systems in time. IFBHP (Instruction Flow Based Hybrid Prediction), a mechanism based on branch prediction and hybrid learning is proposed. IFBHP predicts memory accessing instruction streams based on branch prediction. Then it uses hybrid learning mechanisms to predict the address of every memory accessing instruction ,and write the address into a dedicated queue. IFBHP controls sending and writing time of prefetching requests by estimating the time of instruction blocks coming in pipeline. The evaluations prove that IFBHP reduced cache misses of read operations by 31.58%, 28.85%, 17.85%, 11.48% respectively comparing with STeMS (Spatio-Temporal Memory Streaming), ISB++ (Irregular Stream Buffer++), SANGAM, IPCP (Instruction Pointer Classifier based spatial Prefetching). While, IFBHP reduced cache misses of write operations by 31.58%, 28.85%, 17.85%, 11.48% respectively.

Key words: cache prefetching; branch prediction; temporal address correlation; strided access; instruction flow

Foundation Item(s): National Science and Technology Major Projects of Hegaoji (No. 2014ZX01029101-002); Strategic Priority Research Program of Chinese Academy of Science (No.XDA-Y01-01)

1 引言

内存墙[1]指处理器的性能提升速度远大于内存性能的提升速度,导致内存性能严重限制了CPU性能的发挥. 减少访问等待延迟和提高内存访问带宽是增加内存性能的两种基本途径:前者主要采用多级缓存和缓存预取等技术;后者主要采用增加总线位宽和传输

收稿日期:2021-02-23;修回日期:2021-10-27;责任编辑:孙瑶

速率的方法．近年来，提高总线位宽、传输速率和多级缓存等技术的普遍应用导致处理器中的缓存系统占比越来越大，缓存系统设计也越来越复杂，在某些处理器中缓存系统甚至可以使用接近40%的总体面积[2]．因此，高性能处理器必须能够对缓存系统进行高效管理．缓存预取技术是被广泛认可的能够提高缓存系统性能的有效手段，在内存墙问题仍未解决的当下，预取技术是缓存管理相关研究的热点[3]．缓存预取的目的是提前将应用程序可能会用到的数据写入到缓存系统中，降低缓存失效发生的次数，从而减少访问慢速内存引入的性能损失．预取算法通常根据历史信息学习访存地址之间的关联模式，推测预取请求的地址．访存地址之间的关联性源于访存行为具有的时间或空间局部性特征．

早期的缓存预取算法生成预取请求需要特定的触发条件，通常在缓存失效发生时进行预取．这些预取算法以引起缓存失效的访存地址为基础预取若干连续的缓存行[4]，或者预取若干符合步长模式的缓存行[5]．缓存预取算法的近期研究，一方面使得预取请求生成变得更加积极，不再需要特定的触发条件；另一方面能够学习更多、更复杂的访存行为模式[6]．这些基于模式的经验主义算法只针对符合一定特性的程序有效，不能被广泛应用．同时这些算法在缓存系统中学习地址之间的关联模式，容易受到乱序执行的干扰．

最新的缓存预取算法研究使用混合多种模式的学习算法处理复杂的应用场景．STeMS（Spatio-Temporal Memory Streaming）算法[6]充分利用了空间关联和时间关联各自的优势，提出一种能够还原原始访问序列的预取算法．MISB（Managed Irregular Stream Buffer）算法[7]扩展了ISB（Irregular Stream Buffer）算法[8]，同时提出该算法可以和BOP（Best-Offset hardware Prefetching）算法[9]结合，组成更复杂的预取算法．IPCP（Instruction Pointer Classifier based spatial Prefetching）算法[10]和SANGAM算法[11]都是由3种预取算法组成的混合预测器．

还有一些预取算法使用Lookahead技术[12,13]进行预取请求的生成．Lookahead使用主流水线的空闲周期或者专用的预取子流水生成预取请求，该技术能够适用于行为不规律的访存场景中，相比于其他预取算法具有更广泛的应用前景．其中，B-Fetch[12]使用分支预测技术搭建独立于处理器主流水线外的预取子流水，预测未来程序流．该算法记录每条访存指令相关的架构寄存器，并且能够对架构寄存器值的变化进行学习，使用推测的寄存器值计算访存请求目标地址．该算法能够处理不规则的访存行为，但是该算法不能很好地处理依赖缓存失效（dependent miss）[3]．依赖缓存失效是指连续的缓存失效之间存在某种关联，例如引起后续缓存失效的指令依赖前续缓存失效指令的访存数据．同时，B-Fetch假设每个基本块执行过程中对架构寄存器值的修改是一个固定的常量，但是应用程序中经常对寄存器进行乘/除等复杂操作，因此B-Fetch并不能准确估算架构寄存器的值．

现有预取算法均不能保证预取地址的绝对准确，因此预取请求发送的投机性不能设置过高，否则大量的无效预取请求会浪费缓存系统带宽，污染缓存存储阵列，从而降低处理器性能．预取算法都需要控制预取投机深度，比如B-Fetch使用的路径信任的方式[13]以及其他算法使用的自适应的方式等．这些限制预取深度的方法虽然可以很大程度上减少过度预取的损害，但是这些方法并不能精确地控制预取发送的时机，因此不能避免预取的负面影响．

为了解决B-Fetch等预取算法的不足，本文提出了一种新的基于分支预测技术的预取算法．通过分支预测技术搭建预取子流水，预测程序未来指令流，通过访存地址缓冲（Memory Access Address Buffer，MAAB）获取指令流中访存指令的位置和类型等信息，生成访存指令队列．同时，将处理器执行过的每条访存指令的历史访存地址信息记录在独立的访问历史缓冲（Address History Buffer，AHB）中，学习每条指令的访存模式．通过AHB对访存指令队列中的每一条指令的访问地址进行预测，生成访存地址队列．访存地址队列代表了程序对未来数据的需求．本文算法基于访存地址队列中的数据进行预取，避免无用的预取数据污染缓存系统．

本文提出的指令访存模式学习算法是一种混合算法，通过AHB可以学习时间关联和空间关联等多种模式，能够处理依赖缓存失效．通过预取子流水的投机性确保预取请求生成的提前量．访存地址队列记录已经生成的预取地址．本文参照文献[14]提出的最优原则，提出一种方法精确控制预取请求的发送和写回时机．通过分支预测队列和访存地址队列计算已经生成好的预取请求与当前程序执行位置之间的距离，评估处理器的执行速率，把即将执行的访存指令所对应的预取请求从访存地址队列中读出，发送给缓存系统．预取请求完成后不会立刻写回缓存系统，而是停留在预取队列中，直到确信不会将缓存系统中的有益数据踢出．

本文的主要贡献如下：

（1）本文提出了一种新的控制预取深度和预取时机的方法．该方法通过分支预测队列限制预取子流水的深度，并通过该队列计算预取请求所属指令块和CPU主流水当前执行位置之间的距离，并通过评估处理器的执行速率，设置提前量，控制预取请求的发出和写回，在保证预取地址计算准确性的同时，可以更准确地操纵预取请求的发送和写回过程，最大限度减少预取对缓存系统的负面影响．

(2)本文提出了基于指令流混合模式学习的缓存预取算法(Instruction Flow Based Hybrid Prediction, IFBHP). 通过分支预测技术推测程序未来的指令流,并识别指令流中的访存指令. 为每条访存指令独立分配访问地址缓冲记录过往信息,学习地址关联模式并计算预取请求地址. 该方法不依赖于特定分支预测算法,能够识别多种不同的访存地址关联模式,具有良好的可扩展性.

(3)本文使用GEM5模拟器搭建了预取算法的原型系统,通过在SPEC CPU2006程序集中实验评估,验证了IFBHP缓存预取算法的有效性. 同时实验证明了在分支预测和访存地址模式两者均充分学习的情况下,本文算法预测出的预取地址准确性优于现有算法.

2 相关工作

缓存预取算法记录访存地址历史信息,总结地址之间的关联模式,并依此生成预取请求发送到缓存系统. 依据地址序列间的关联模式可以将预取算法分为3类:步长模式、时间关联模式和空间关联模式[15].

如果一段访存地址序列中相邻地址间的差值是一个或者多个固定值,这个访存序列的关联模式即为步长模式,例如访存地址序列$\{M, M+S, M+2S, \cdots\}$. 步长模式是最常见的地址关联模式. 程序中的矩阵操作或者指针操作产生的地址序列大都属于步长模式. 最初的步长算法[16]仅能处理差值为一个固定值的地址序列,近期研究丰富了差值的识别范畴,能够支持多种差值[17]或者可变差值[18]的步长模式. 偏移预取算法是对传统步长算法的创新. 偏移预取算法的核心是确定一个地址偏移,用当前访存地址$\{M\}$加上这个偏移$\{O\}$生成预取地址$\{M+O\}$. 偏移$\{O\}$并不是一个固定值,可以根据场景自适应地变化. SP(Sandbox Prefetching)[19]算法评估特定范围内每个偏移的准确性,根据评分选定偏移计算预取请求地址. BOP算法[9]改进了偏移的评分规则,用时效性取代准确性作为评分依据. MLOP(Multi-Lookahead Offset Prefetching)算法[20]对BOP算法进行了扩展,在学习周期对各个偏移进行评分时区分不同的超前等级,从每一个超前等级中选择最高评分的偏移计算预取地址. 步长模式是一种非常简单且规律的地址关联,实际应用场景中往往存在大量不规则的地址序列,步长预取算法遇到这种场景就会失效.

如果一段访存地址序列中某个子序列以相同的顺序多次出现,这个子序列的关联模式即为时间关联模式. 例如访存地址子序列$\{A, B, C, D\}$在访存历史中多次出现,这4个地址的连续访问行为就属于时间关联模式. 当地址A再次出现时,后续的地址序列大概率是$\{B, C, D\}$,因此可以对$\{B, C, D\}$进行预取. 时间关联预取算法[21]通常将缓存失效地址按照顺序保存在特殊失效缓冲中,因此若干连续地址间的时间关联很自然地记录在该缓冲中. 生成预取请求时,使用当前失效地址查询失效缓冲,获得以失效地址为首的一段子序列,作为预取候选项. TSSM(Temporal Streaming of Shared Memory)算法[22]减少了对索引表和失效缓冲的访问次数,降低了预取算法的额外开销. ISB算法强化了不规律地址序列的处理能力. ISB算法新定义了结构地址空间(Structural Address Space, SAS). 结构地址和物理地址的互相转化通过映射表进行. 引入结构地址空间的目的是将物理地址空间中不规律的地址序列转化为结构地址空间中的连续地址序列. 时间关联预取算法能够处理不规律的访存行为,也能够处理依赖缓存失效. 时间关联预取算法的缺点包括占用较多的存储资源,容易造成过度预取,以及不能处理强制性失效(compulsory miss).

如果一段地址偏移序列在内存的不同区间多次出现,这段地址偏移序列的关联模式即为空间关联模式,例如区间M中存在偏移为$\{A, B, C, D\}$的地址序列,如果在其他区间中也存在相同的偏移序列,此偏移序列就属于空间关联模式. 空间关联预取算法[23]通常将整个地址空间按照固定尺寸分割为独立的子区间,在每个区间中学习地址偏移间的关联. 在某个区间中识别到的地址关联,可以应用到该区间以及其他区间. AD/DC预取算法[24]在传统空间预取算法的基础上使用地址间的差值替代完整地址,减少了存储资源开销. Bingo算法[25]记录多张模式历史表,每张表的检索历史长度不同,预取请求生成时选择历史长度最长的模式表中的数据. Bingo算法能够将不同的历史表整合在一起,进一步减少资源开销. 空间关联预取算法资源开销比时间关联算法少,同时可以减少强制性失效,缺点在于不能处理依赖缓存失效.

以上3类算法分别学习了不同的地址关联. 预取算法还可以混合多种地址关联模式,生成更复杂的算法. STeMS算法充分利用了空间关联和时间关联的优点,一方面参照空间关联模式在不同区间内学习访存模式,另一方面将不同区间的触发地址按照顺序记录下来. MISB算法优化了ISB算法,可以更高效地管理存储结构,减少额外的访存开销. 同时MISB算法可以和BOP算法结合,由MISB算法预测不规律的访存模式,BOP算法预测规律的访存模式. IPCP算法和SANGAM算法都是由3种预取算法组成的混合预测器,包含了规律模式学习和不规律模式学习等不同的算法. IPCP由固定步长预测器、复杂步长预测器和全局流预测器组成. IPCP的复杂步长预测器为每一条访存指令生成签名,使用该签名索引共享的步长预测表获取下一次步

长.IPCP的全局流预测器统计不同区域内的访问向量,识别聚集访问行为.

除了基于访存地址模式学习的预取算法外,还有一些借助预取流水的算法.文献[26]提出在多核处理器场景中使用其他核协助生成预取请求的方法.R3-DLA(“Reduce,Reuse,Recycle”-Decoupled Look-Ahead)算法[27]不仅可以利用多核DLA框架传递预取信息,还可以传输分支预测信息和值预测信息等,加速主线程的执行.Runahead Execution技术[28]不需要多个处理器核,而是使用当前核的空闲周期生成预取请求.上述方法需要空闲核/空闲周期,在负载比较重的场景中很难应用.Lookahead技术[12,13]在处理器主流水外搭建轻量级的预取子流水,通过预取子流水生成预取请求.Lookahead技术的缺点是不能很好地控制预取子流水的投机性,预取请求的发送时机可能会过早或者过晚.并且Lookahead技术很难处理依赖缓存失效[3].本文提出的IFBHP算法借鉴Lookahead技术构建预取子流水.IFBHP算法将访存指令的历史信息记录到AHB中,能够识别时间关联模式和步长模式等多种模式.IFBHP算法一方面增大了预取子流水的预测带宽,确保预取请求及时生成;另一方面通过指令块个数计算预取请求和程序当前执行位置之间的距离,将生成的预取请求在最恰当的时刻发送到缓存系统.

现有预取算法研究大都只重视预取请求生成的准确性,很少涉及预取请求之间的顺序,这就意味着现有算法很难确定预取请求间的优先级.STeMS算法注意到了这个问题,保存了失效地址的顺序,当同时处理多个预取地址序列时通过次序信息排序各个预取请求.本文算法根据访存指令的PC确定其在程序中的先后次序,并依次排序各条访存指令的预取请求.本文实验环节证明了预取请求的发送存在最佳时间窗口,而当前预取算法均不能确保在此窗口内发送预取请求.

3　研究背景

本文算法基于处理器中的分支预测单元进行扩展,记录访存指令信息的MAAB借鉴了分支预测器(Branch Target Buffer,BTB)中的设计思想.下文对这方面的内容进行介绍.

分支预测技术根据历史信息预测分支指令的行为,降低分支刷新造成的性能损耗.分支预测分为地址预测和方向预测,分别预测分支指令的目标地址和跳转与否.分支方向预测将分支指令间的关联总结为模式,采用模式识别的手段预测分支行为.

图1是包含2比特饱和计数器的BTB结构示意图[29].BTB是分支预测器中的主要组成部分,它记录流水线中执行过的分支指令的位置和类型等信息,通过分支指令的PC进行检索.BTB主要解决两个问题:(1)判断分支指令是否存在;(2)确定分支指令的目标地址.本文进行访存指令预测时也要解决两个类似的问题.因此本文参照BTB结构和功能实现MAAB,记录访存指令的PC、访存地址以及指令类型等信息.

分支指令PC	分支指令跳转地址	饱和计数器
0x133EFO	0x134DFC	1
…	…	…
0x133396	0x135BCD	3

图1　BTB结构

4　缓存预取算法

缓存预取(IFBHP)算法结合分支预测和地址关联模式学习等技术对程序的未来访存地址序列进行预测,通过此序列生成预取请求.访存地址序列的预测分为2个步骤:(1)预测未来访存指令序列;(2)对访存指令序列中的指令逐一进行地址预测,生成访存地址序列.

首先,本文将处理器执行过的访存指令信息记录在MAAB中.MAAB的设计仿照BTB,记录访存指令的PC、访存地址和指令类型等信息.通过检索MAAB可以确定某段指令流中访存指令是否存在,结合分支预测器对指令流的预测,即可以预测未来访存指令序列.

然后,本文将访存指令的过往地址信息独立地记录在访存历史缓冲AHB中.通过AHB中记录的历史信息,本文可以识别多种常见的地址关联模式,根据访存指令类型使用对应的模式预测器进行预取地址计算.AHB根据指令提交的顺序记录访存地址信息,不受处理器乱序执行机制的影响.通过访存模式的学习,本文对访存指令序列中每条指令的目标地址逐一进行预测,生成访存地址序列.

访存地址序列保存已经生成的预取请求,但是预取请求并不会立即发送给缓存系统,而是先暂存在该队列中.本文按照文献[14]提出的最优化规则控制预取请求的发送和写回时机.仅当访存指令将要进入流水线时才会将对应的预取请求发出,最大限度地避免由于预取导致缓存中的有效数据被踢出.

本文算法的总体架构如图2所示,在原流水线外新增了预取子流水,该子流水分为访存指令预测模块、访存地址预测模块和预取请求生成模块3个部分.访存指令预测模块通过预测循环将预测的指令流信息存入分支预测队列,通过该指令流信息查询MAAB,写入访存指令序列;访存地址预测模块通过AHB学习地址关联模式,对访存指令序列中的指令逐个进行地

址预测，写入访存地址序列；预取请求生成模块在特定时机查询访存地址序列，控制预取请求的发送和写回过程.

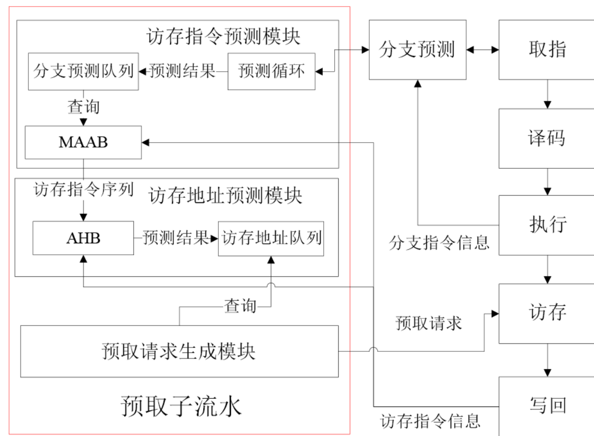


图2 IFBHP算法框架

提前流水设计（ahead pipeline）[30]为预取子流水的预测准确性提供了依据. 当主流水线中的取指单元因缓存失效等情形停止工作时，预取子流水仍然可以继续工作. 仅当分支预测队列没有空闲项时，预取子流水才会停止工作. 预取子流水的预测速率（128字节/周期）是主流水线取指速率（64字节/周期）的两倍，因此通常情况下预取子流水的执行会远远领先于主流水线，确保预取请求生成的及时性. 分支预测队列包含读/写指针和提交指针，执行完毕的指令将从队列中删除.

下文会详细描述访存指令预测模块、访存地址预测模块和预取请求生成模块的设计细节.

4.1 访存指令预测模块

访存指令预测模块的设计目标是预测程序未来的访存指令流. 访存指令预测模块包含预测循环、MAAB和分支预测队列3个子模块. 其中，预测循环是最关键的一个子模块，对程序未来的指令流进行预测. 预测循环的预测结果以指令块为粒度写入分支预测队列. 分支预测队列保存的是完整的指令流，在此基础上查询MAAB可以得到访存指令序列.

4.1.1 预测循环

预测循环本质上是一个分支预测器，按照固定的带宽进行预测，并根据预测结果继续预取下去，直到分支预测队列满. 预测循环包含提前预测地址、提前预测分支目标缓存（Ahead Branch Target Buffer，ABTB）和提前预测历史3部分. 提前预测地址负责管理当前预测地址，表示为<pc, op>，其中pc代表当前周期的预测地址，op代表当前周期对预测地址的操作. 提前预测历史是预取子流水新增的分支历史，在预测过程中投机地进行更新. ABTB采用组相联结构，512组、4路，共4 096项，使用PLRU替换策略. 每一项的结构定义为<Valid, Tag, Offset, Instlen, TargetAddr, BranchType>. 其中，Valid为有效位；Tag为标签；Offset为指令块内偏移；Instlen为指令长度，用于计算指令的首字节地址；TargetAddr为跳转目标地址；BranchType为分支类型.

图3是预测循环的工作流程，预测循环以提前预测地址为起点开始预测. 在处理器初始化或者发生刷新时，取指单元将最新的地址发送给预测循环，用于提前预测地址的更新. 工作流程第一步检索ABTB，检测以预测地址起始的对齐的128字节窗口（分为8个16字节窗口独立预测）中是否存在分支指令. 若存在分支指令，则通过分支预测单元查询分支指令的预测结果；若预测窗口内不存在分支指令或者存在的分支指令均不跳转，则用128字节对齐加一的形式更新提前预测地址；若存在跳转的分支，则使用分支指令目标地址对提前预测地址进行更新.

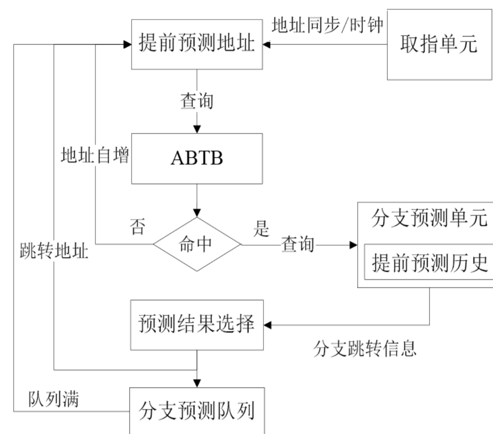


图3 预测循环工作流程

预测循环中用分支指令的末位字节地址当作其标识，而分支预测单元中的BTB以分支指令的首位字节当作其标识. ABTB可以判断分支指令存在与否，同时可以通过指令长度信息进行指令首尾地址的转换.

ABTB将分支指令的地址划分为高位的标签和低位的偏移. ABTB在进行检索时，使用预测窗口的地址进行范围匹配. 128字节的预测窗口拆分为8个16字节的子窗口进行并行检索. 每个16字节子窗口检索ABTB时使用窗口起始地址索引ABTB中的一组，读取该组中的4条分支指令信息. 范围匹配首先对比窗口起始地址高位和分支指令的标签，若标签相等则继续对比分支指令的偏移和窗口地址低位，仅当偏移大于等于地址低位时才算范围匹配命中. ABTB的设计支持16字节预测窗口中最多命中4条不同分支指令.

ABTB中每一条命中的分支指令会检索分支预测单元，各条指令的预测结果将汇总在一起进行预测结果优先级选择. 优先级逻辑将偏移最小的跳转分支当

作整个窗口的最终预测结果.若预测窗口内存在跳转的分支指令,则以预测窗口地址为起始,以分支指令末位字节为终点构造指令块信息,写入分支预测队列.如果预测窗口内不存在分支或者分支均不跳转,则以预测窗口的末位字节作为其终点构造指令块.如果某个16字节预测窗口内识别到了跳转的分支,则后续16字节窗口的预测结果不必写入分支预测队列.

4.1.2　分支预测队列

分支预测队列以64字节指令块为粒度存储预测循环输出的未来指令流.分支预测队列是一个循环队列,共有400项.其结构定义为<Valid, Taken, LineAddr, BeginOffset, EndOffset>.其中,Valid代表有效位;Taken代表跳转位;LineAddr代表指令块地址;BeginOffset代表起始偏移;EndOffset代表终点偏移.指令块地址和起始偏移/终点偏移划定了指令块内有效字段的范畴.

分支预测队列的长度限制了预测循环的投机深度.分支预测队列中每一项代表一个取指指令块,每一个指令块根据CPU主流水线带宽一般需要多个周期才能处理完毕.即便考虑到预取请求自身的处理时延,分支预测队列采用400项也是足够的.

预测循环的预测带宽是128字节,每次最多分配两个分支预测队列项.指令块选择64字节宽度的目的是匹配取指单元的处理宽度,简化读指针的维护逻辑.如图4所示,分支预测队列包含读指针、写指针和提交指针.指令块从队列中删除的条件是其中所有指令均已提交,然后提交指针加一.读指针随取指单元的执行自增.预测循环每次最多分配两个分支预测队列项.写指针根据分配个数进行更新.读指针和提交指针之间是已经进入流水线的指令块;读指针和写指针之间是将要进入流水线的指令块.通过指令块指针与读指针间的距离可以预估指令块进入流水线的时间.

如图4所示,指令块M中存在多条访存指令,其地址分别为A,B,C,存储在访存地址队列中.每个访存地址队列项中都包含分支预测队列的指针,因此每一个访存地址都能够找到对应的指令块.通过计算读指针和指令块M间的差值,可以估算出指令块M进入流水线的时间,进而适时发出指令块M中包含的预取候选项.

若主流水线发生刷新,分支预测队列的读写指针会进行回滚,访存地址队列也会相应地同时更新,然后预测循环会重新填写两个队列的数据.在本文算法中预取请求的生成和发送过程是互相独立的,访存地址队列中记录的已经生成好的预取请求在等候发送期间可以进行数据更新.因此,虽然预测循环的投机深度很高,但是预取的准确性是可以保证的.

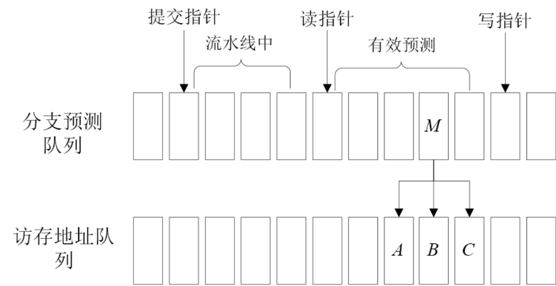


图4　分支预测队列

刷新发生时分支预测队列读写指针的更新流程如图5所示.当分支指令造成处理器刷新时,执行单元会反馈该指令相应的分支预测队列指针.分支预测队列的读写指针根据反馈的指针位置进行更新.

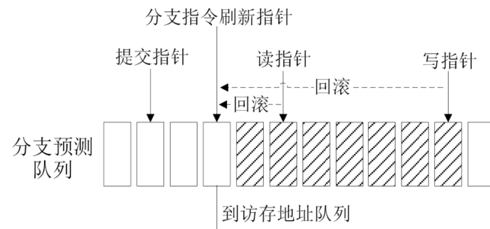


图5　分支预测队列刷新

4.1.3　MAAB

MAAB保存执行过的访存指令的信息.MAAB存储结构有4 096个表项,根据主流水线写回单元反馈的信息,保存访存指令的PC,指令类型以及访存地址等信息.其结构定义为<PC, PhyAddr, InstType>.其中,PC代表访存指令的地址;PhyAddr代表上一次执行访问的物理地址;InstType代表指令类型.InstType ∈{Normal, Stride, Temporal},其中Normal代表目标物理地址不变的普通访存指令,Stride代表满足步长模式的访存指令,Temporal代表满足时间关联模式的访存指令.

预测循环分配分支预测队列项时会随即进行MAAB的检索.检索时用新增指令块的起始地址查询MAAB,查找指令块中的访存指令,并将找到的访存指令按顺序发送给访存地址预测模块.MAAB在检索过程中通过访存指令的PC进行范围匹配,匹配过程判定MAAB中记录的访存指令是否在分支预测队列新增指令块的区间范围内.

同一个指令块中可能含有多条访存指令,因此每次MAAB的检索可能会匹配多项.这些访存指令的信息将按照指令流中的顺序组成访存指令序列,发送给访存地址预测模块.

4.2　访存地址预测模块

访存地址预测模块完成预取请求的生成过程,根据接收到的访存指令序列,为序列中的每一条访存指

令进行地址预测，其结果写入访存地址队列. 访存指令的预测需要借助AHB学习到的访存模式. 访存地址队列中存储的是预取的候选项，预取请求不会立即发送到缓存系统，队列中的数据会在恰当的时刻被预取请求生成模块使用.

4.2.1 AHB

AHB接收主流水线写回单元发送的访存地址，并根据记录的历史地址学习访存指令的模式. AHB由访存指令PC进行索引，共4 096项，与MAAB项一一对应. 为了减少资源消耗，访存模式的学习和记录是独立的两个阵列. AHB中每一项只包含4个地址，记录已经学习好的访存模式，访存模式学习由额外的活跃访问历史缓冲(Active Address History Buffer，AAHB)完成. AAHB是附属于AHB的子阵列，由访存指令PC索引，共256项，每一项中包含12个地址. AAHB追踪流水线中近期执行过的访存指令，记录过往12次访问的地址信息，并学习其中的地址关联模式. 当AAHB识别到访存指令的模式之后，将模式信息记录到AHB中.

本文算法基于过往历史信息可以学习多种访存模式，属于混合预测算法. 步长模式需要计算过往访问地址之间的差值，时间关联模式需要从过往地址序列中查询重复出现的子序列，这些信息都可以通过AHB获得.

区别于传统的地址关联模式学习方法，本文算法进行访存地址预测时必须要明确整个地址关联序列的长度. 假设访存指令M的地址序列属于长度为4的时间关联模式$\{A,B,C,D\}$，如图6所示，包含访存指令M的指令块在分支预测队列中重复出现. 为了能够精确预测访存指令M第五次执行的地址为A，必须获取该地址序列的长度. 因此地址关联序列的准确长度是本文算法必须要学习的内容.

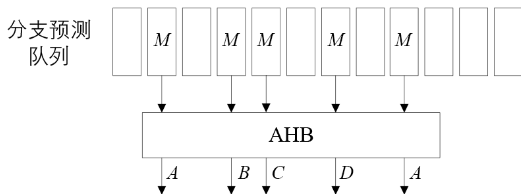


图6 基于AHB的预测依赖序列长度信息

本文算法实现了步长模式预测器和时间关联模式预测器，下文对这两种预测器的设计进行简单介绍.

4.2.1.1 步长模式预测器

步长模式是最常见的访存地址关联模式. 本文设计的步长预测器借助AAHB中记录的访存指令历史信息，使用额外存储记录步长模式相关信息，其结构定义为

<Mode, Stride, BaseAddr, LastAddr, CurCounter, MaxCounter, Direction, ChangeBaseMode, ChangeBaseStride, ChangeBaseDirection>.

本文算法在传统步长模式算法[15]的基础上扩展了当前模式(mode)、首地址(baseAddr)、当前计数器(CurCounter)以及最大计数器(MaxCouter)等内容. 这些新增内容用于记录步长地址序列的起始/终点以及序列的长度. 此外，针对首地址(BaseAddr)发生变化的情形，还增加了ChangeBaseMode模式和相应的步长信息.

实验环节对SPEC程序的分析发现在程序存在形如$\{M,M+K,M+2K,N,N+K,N+2K,\cdots\}$的地址序列(其中$M$和$N$是不同的BaseAddr，$K$是Stride). 此地址序列具有以下特点：步长和序列长度是固定的，但是首地址是变化的(从M到N)，同时每次首地址变化的差值是固定的. 本文步长预测器支持对这种地址序列的学习.

空间关联模式可以将某个内存区间中学习到的模式应用到其他区间中，减少强制缓存失效. 本文支持首地址可变的步长地址序列，从首地址M切换到首地址N的过程类似于空间关联模式在不同的区间进行切换，因此本文算法也可以减少部分强制缓存失效.

步长预测器的访存地址计算如算法1所示.

算法1 步长预测器的访存地址计算

输入: 步长预测器信息
输出: 预测地址

```
IF (当前模式==步长模式)
   根据编号检索分支预测队列, 获取包含该指令尚未提交的指令块个数N
   IF(BaseAddr可变)
      Num_Cycle=(CurCounter+N+1)/(MaxCoutner+1)
      Num=(CurCounter+N+1)%(MaxCoutner+1)
      根据Num_Cycle调整LastAddr
      预测地址=LastAddr+Direction*Num*Stride
   ELSEIF(MaxCoutner !=0)
      Num=(CurCounter+N+1)%(MaxCoutner+1)
      预测地址=BaseAddr+Direction*Num*Stride
   ELSE
      Num=1+N
      预测地址=LastAddr+Direction*Num*Stride
   ENDIF
ELSE
   预测地址=LastAddr
ENDIF
```

4.2.1.2 时间关联模式预测器

时间关联模式也是比较常见的访存地址关联模式. 通过AAHB中记录的访存历史信息进行时间关联模式的学习. 本文设定某个地址子序列按照同样的次序出现3次才会识别为时间关联模式. AAHB保存的地

址数为12,因此本文支持长度不大于4的时间关联序列.

除此之外,本文算法还支持形如$\{A, B, C, A+K, B+K, C+K, A+2K, B+2K, C+2K, \cdots\}$这样的步长——时间关联行为模式.

时间关联模式的识别如算法2所示.

算法2 时间关联行为识别

输入: AAHB$[M-1:0]$中记录的过往M次访存地址

输出: 时间关联模式信息

IF (AAHB$[3n-1:0]$匹配$\{A_1, A_2, \cdots, A_n, A_1, A_2, \cdots, A_n, A_1, A_2, \cdots, A_n\}$)

  时间关联模式=1

  时间关联序列长度=n

  时间关联序列=AAHB$[n-1:0]$

ELSEIF(AAHB$[3n-1:0]$ 匹配$\{A_1, A_2, \cdots, A_n, A_1+k, A_2+k, \cdots, A_n+k, A_1+2k, A_2+2k, \cdots, A_n+2k\}$)

  时间关联模式=1

  步长=k

  时间关联序列长度=n

  时间关联序列=AAHB$[n-1:0]$

ENDIF

IPCP算法的复杂步长预测器中提到了一种差值-时间关联模式,即具有时间关联特性的不是访存地址本身,而是访存地址之间的差值,例如差值序列$\{1, 2, 1, 2, 1, 2, \cdots\}$. 通过AAHB中记录的完整访存地址是可以计算出地址差值历史的. 本文在AAHB的基础上实现了对差值-时间关联模式的识别. 差值序列学习的最大长度仍然是4.

4.2.2 访存地址队列

访存地址队列存储已经生成的预取地址. 本文假设64字节指令块中平均包含8条访存指令,因此访存地址队列共有3 200项. 每一项的结构定义为

< Valid, InstLineAddr, MemPhysAddr, MemLen, InstQueueIndex >.

其中,Valid代表有效位;InstLineAddr代表访存指令PC;MemPhysAddr代表访存物理地址;MemLen代表访存位宽,用于识别访存数据是否跨缓存行;InstQueueIndex代表所属指令块的分支预测队列索引. 与分支预测队列相同,访存地址队列中也实现了读指针、写指针和提交指针,当流水线刷新时可以进行指针回滚.

4.3 预取请求生成模块

由于预取子流水通常条件下会大幅领先于处理器主流水,访存地址队列中记录的已经生成的预取地址大概率在很久以后才会被程序使用. 预取请求最恰当的发送时机是预取请求对应的指令即将进入流水线的时候.

文献[14]研究了理想情况下的缓存预取应当满足的规则. 其中,规则3要求预取请求不能发送得过早,不能将访存序列中更靠前的地址数据从缓存中替换出来. 规则4要求预取请求不能发送得过晚,应当在它能够发出的第一时间发送出去. 规则3结合规则4约束了预取请求发送到缓存系统的时机. 本文算法在控制预取请求的发送时兼顾了规则3和规则4的要求,一方面及时将预取请求发送到缓存系统,另一方面尽可能避免预取数据对当前缓存内容的损害.

预取请求的发送过程如图7所示. 以一级数据缓存的访问作为预取发送过程的触发条件. 新的缓存访问发生时,预取请求生成模块第一步查询分支预测队列,获取其读指针和写指针. 读指针与写指针之间的指令块均为尚未被处理器读取的,根据指令块与读指针间的距离可以评判其进入流水线的时间. 本文在分支预测队列读指针位置上添加特定的预取提前量,当作预取的开始位置,每次选择连续的N个指令块发送其中的预取请求. 预取请求生成模块将分支预测队列写指针与读指针间的距离作为有效指令块的个数,仅当该个数超过预取提前量和N之和时才会发送预取请求.

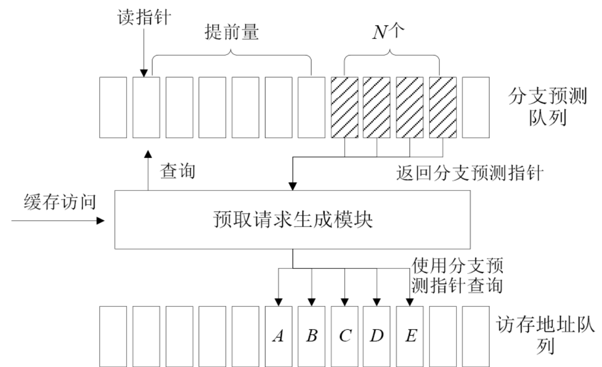


图7 预取请求发送过程

预取请求生成模块从分支预测队列中获取到预取指令块的指针后,通过这些指针查询访存地址队列,获取这些指令块内包含的预取候选项,然后将所有预取请求发送给缓存系统.

预取请求在缓存系统处理完毕后立刻将其数据写回缓存阵列有可能会将缓存中的有益数据踢出. 本文添加了预取队列,暂存已经拿到数据预取请求. 预取队列是一个先进先出的循环队列,共有50项. 完成的预取请求需要在队列中等候,直到确信缓存中有合适的写入位置.

预取请求写回等待过程如图8所示. 预取请求M发出后,需要等待缓存系统反馈数据. 预取请求M拿到数据后首先进入预取队列. 当其到达预取队列首位时,根据预取请求的分支预测队列指针检索分支预测队列

和访存地址队列,查询在M所属的指令块之前是否存在和它地址同组的其他预取请求.若没有和M地址同组的其他请求,预取请求M可以立即写回缓存阵列;若存在和M地址同组的请求,则预取请求M必须进行额外的写回判定.如图8所示,经查询发现,访存地址队列中4个预取的地址A,B,C,D和预取请求M同组,假设缓存阵列只有4路,预取请求M替换A,B,C,D中的任意一个都会违反文献[14]中的规则3.因此需要查询第一个同组预取请求A在分支预测队列中的指针L,仅当分支预测队列的提交指针大于L时,才可以将预取请求M写回缓存阵列.

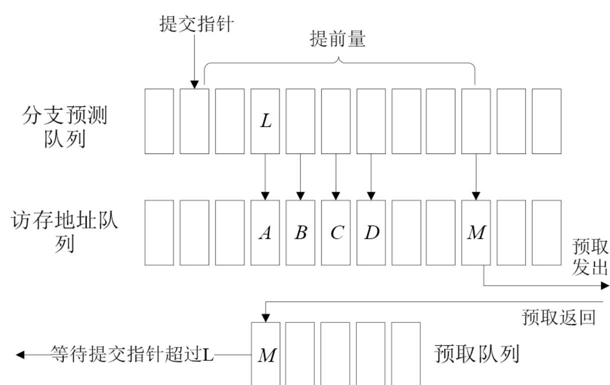


图8 预取请求写回等待过程

5 实验

本节通过实验验证IFBHP算法的有效性:首先介绍实验环境和测试用例,然后比较IFBHP算法和其他预取算法的性能.

5.1 实验设置

本文使用GEM5模拟器[31]作为实验平台,采用alpha指令集.实验环节需要对主流水线的功能进行扩展,因此选择细粒度的DerivO3CPU模型.算法中使用到的分支预测算法为TAGE-L. GEM5的基本配置如表1所示.其中比较重要的配置参数包括缓存容量以及其处理时延,取指和解码的带宽等.这些参数决定了CPU的执行速度.本文算法预取提前量的选择与这些参数是密切相关的.本文期望预取子流水能够大幅领先于处理器主流水,从而保证访存地址队列和分支预测队列中有充足的预测信息.

实验选择SPEC 2006程序作为测试集.GEM5运行完整的SPEC 2006测试程序可能需要几个月的时间,通用的做法是从程序中的Simulation Point开始运行.本文根据文献[32]中提供的检查点数据进行仿真.SPEC 2006整型和浮点程序的检查点如表2、表3所示.

实验从上述测试程序的检查点起始运行1 000万条指令作为测试片段.实验的对比算法为STeMS算法[6]、IPCP算法[10]、ISB++算法[7]和SANGAM算法[11].

表1 GEM5配置

参数	配置值
cpu-clock/GHz	3.0
cpu-type	DerivO3CPU
bp-type	TAGE_L
l1d_size/KB	32
l1d_assoc	8
l1d_tag_latency	2
l1d_data_latency	2
l1d_mshrs	20
l2_size/KB	256
l2_assoc	8
l2_tag_latency	20
l2_data_latency	20
mem_size/MB	4 096
fetchBufferSiz/B	64
decodeWidth	8

表2 整型程序检查点

CINT06	Simpoint点(亿条指令)
401.bzip2	3 792
445.gobmk	530
456.hmmer	3 313
458.sjeng	14 714
464.h264ref	5 271
471.omnetpp	3 772
473.astar	3 287

表3 浮点程序检查点

CFP06	Simpoint点(亿条指令)
410.bwaves	20 179
450.soplex	1 807
459.GemsFDTD	1 842
470.lbm	615

IPCP和SANGAM和本文算法类似,都是能够识别多种访存模式的混合预测器.STeMS算法也是一种混合预取算法,结合了时间关联算法和空间算法的优点.ISB++算法是文献[7]提出的理想化的ISB算法,忽略了访问内存数据引入的额外开销.本文实验对比不同算法在一级数据缓存上发生缓存失效的次数和处理器的IPC.

5.2 实验评估

本节首先对IFBHP算法中使用到的提前量的取值进行评估,然后对比IFBHP算法和其他算法的性能,最后说明IFBHP算法的开销.

5.2.1 提前量的评估

预取请求生成模块发送预取请求时会在分支预测队列读指针上添加提前量当作预取指令块的开始位

置. 本节实验通过设置不同的预取提前量执行测试程序,评估其对本文算法性能的影响.

从图9的实验数据可以得出,大多数测试程序中缓存失效次数是随着预取提前量的增加而逐渐下降的. 这是因为预取请求的数据只有及时写入到缓存中,才能真正减少缓存失效的次数. 图9的实验还可以得出,缓存失效次数下降的趋势并不是一直维持的,随着预取提前量的持续增加,缓存失效的次数可能微弱地增加. 这一现象说明预取数据过早写入缓存可能会导致缓存内的有益数据被替换出去,进而造成缓存失效次数的增加. 上述实验表明,预取请求的发送是有一个最佳时期的,本文算法能够确保大多数预取请求位于这个窗口期内. 通过实验分析,本文算法的预取提前量设置为120,每次预取4个指令块中.

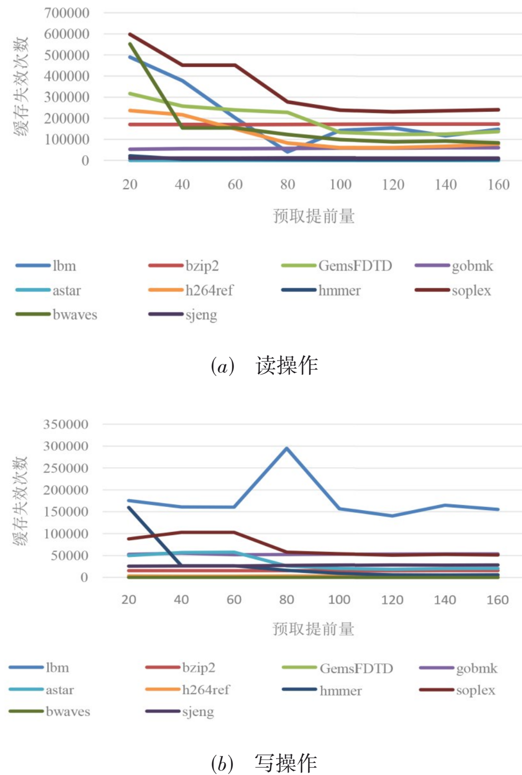
5.2.2 性能对比

本节实验对比了STeMS算法、IPCP算法、ISB++算法、SANGAM算法和本文提出的IFBHP算法的性能. 使用一级数据缓存失效次数和IPC作为对比算法性能的标准.

本文实验分别统计了一级数据缓存读/写操作的缓存失效次数. 如图10和图11所示,IFBHP算法相比STeMS算法、ISB++算法、SANGAM算法、IPCP算法读操作的缓存失效次数分别平均减少50.43%,43.42%,32.05%,17.75%;IFBHP算法相比STeMS算法、ISB++算法、SANGAM算法、IPCP算法写操作缓存失效次数分别平均减少31.58%,28.85%,17.85%,11.48%. 图10和图11从各个预取算法对缓存失效的覆盖情况来对比不同算法的性能. 从图中可以看出,SANGAM算法、IPCP算法和IFBHP算法等支持多种访存模式的混合预测器相比STeMS算法和ISB++算法能够覆盖更多的缓存失效. 预取算法的缓存失效覆盖率是和算法的投机性相关的. 增加算法的投机性可以增加缓存失效覆盖率,但是过多的无用预取请求会浪费缓存系统的带宽,反而可能导致处理器性能下降. 不同预取算法的访存开销分析见5.2.3节.

本文统计了不同预取算法运行SPEC程序的IPC,如图12所示. 本文算法相比STeMS算法、ISB++算法、SANGAM算法、IPCP算法IPC分别提升57.78%,63.31%,11.05%,9.7%.

IFBHP算法大多数情况下均优于其他对比算法,只在sjeng,bzip2和gobmk这3个测试程序中表现较差. 本文统计了IFBHP算法在各个测试程序中发生访存地址预测错误的次数,如表格4所示. 从表格4的实验数据可以得出,大多数SPEC程序中IFBHP算法预测错误的比例是很小的,只有sjeng,bzip2和gobmk这3个程序中预测错误的比例比较高,导致算法性能在这3个



(a)　读操作

(b)　写操作

图9　不同预取提前量的缓存失效次数

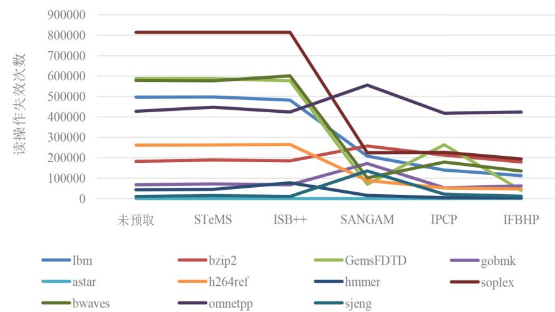


图10　不同SPEC程序读操作失效次数对比

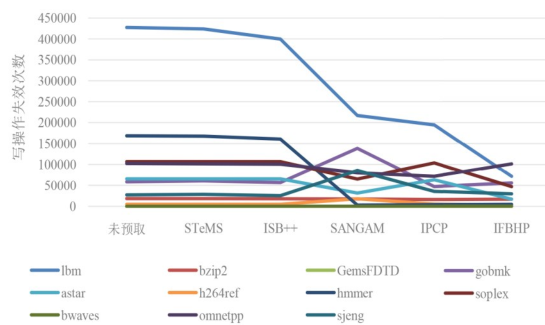


图11　不同SPEC程序写操作失效次数对比

SPEC程序上比较差. 进一步分析发现,这3个SPEC程序中时间关联地址序列并不稳定,比如地址序列第一

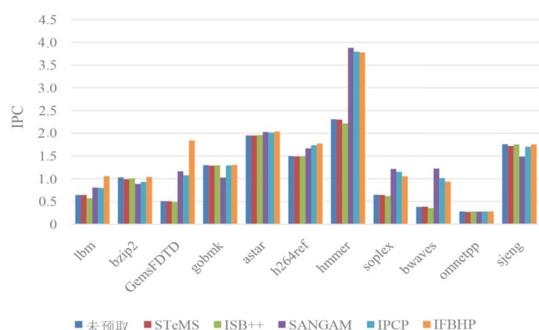


图12 不同预取算法IPC对比

次出现是$\{A,B,C,D\}$,而下一次出现则变为$\{A,B,C,E\}$.识别此类序列需要添加模糊匹配机制,但是本文算法中对时间关联地址模式的学习比较严格,因此不能处理此类应用场景.

5.2.3 算法开销

IFBHP的存储阵列都保存在芯片内部,访问这些存储阵列不会引入额外的访存开销.IFBHP算法的存储开销主要集中在ABTB、MAAB、访存地址队列和分支预测队列上.ABTB有4 096个表项,每个表项96 bit,耗费48 KB的存储资源.MAAB+AHB有4 096个表项.每个表项中包含了71 bit的访存指令信息,68 bit的步长预测器和133 bit的时间关联预测器,耗费136 KB的存储资源.AAHB有512项,每一项510 bit,耗费32 KB的存储资源.分支预测队列有400项,每一项86 bit,耗费4 KB的存储资源.访存地址队列有3 200项,每一项94 bit,耗费37 KB的存储资源.

表4 访存地址预测准确性

SPEC程序名	访存指令总数	预测错误个数
lbm	2 334 698	24 927
bzip2	4 026 021	1 052 529
GemsFDTD	3 292 953	2001
gobmk	3 549 472	1 352 387
astar	3 228 638	18 273
h264ref	3 080 474	390 284
hmmer	3 745 523	9 507
soplex	2 172 495	238 551
bwaves	2 834 032	1 377 984
sjeng	3 326 382	933 828
omnetpp	5 094 293	821 042

IFBHP算法、IPCP算法以及SANGAM算法等的预取请求发送次数统计如表5所示.表5中不仅统计了发送预取请求的个数(HardPFReg_mshr_misses),还统计了未使用的预取请求个数(unused_prefetches).从表5可以得出,相比其他算法,IFBHP算法发送的预取请求个数较少,同时无用预取请求的占比是最低的.这一方面是因为IFBHP算法对预取地址的预测比较准确;另一方面是因为IFBHP算法只在访存指令即将执行的情况下才会为其发送预取请求,不会产生大量的无用预取.

表5 不同SPEC程序预取请求次数对比

SPEC程序名	ISB++	STeMS	IPCP	SANGAM	IFBHP
lbm	397 188/368 988	319 488/313 724	318 802/15 808	475 628/203 835	438 751/13 857
bzip2	92 438/82 720	126 002/125 514	217 177/202 108	525 734/511 462	5 660/3 678
GemsFDTD	247 002/234 514	2 173/2067	216 398/12 348	222 012/20 102	221 075/815
gobmk	17 797/10 850	24 587/23 467	88 919/58 806	1 914 669/1 822 784	10 375/6 303
astar	8 986/8 522	21/10	8 042/7 179	7 238/2 660	7 727/1 499
h264ref	22 733/18 604	16 663/16 338	59 073/32 577	501 890/463 897	30 944/12 192
hmmer	33 002/28 775	10 294/9 452	39 589/3 400	319 339/210 801	36 006/938
soplex	225 806/213 907	111 005/109 565	215 427/109 997	420 170/299 344	107 668/19 270
bwaves	139 349/138 676	2 917/2 753	178 637/1962	565 553/388 572	148 979/664
sjeng	5 460/2 871	28 349/27 768	66 365/53 647	996 833/940 281	3 763/3 226
omnetpp	128 440/115 483	249 701/247 365	346 879/305 275	2 004 062/1 927 858	9 397/6 173

5.3 实验结论

本文的实验数据说明预取请求进入流水线是有一个最佳时间窗口的.本文算法能够精确控制预取请求进入缓存系统的时机,这是其他预取算法不具备的.预取最佳时间窗口并不是固定的,它是与处理器执行速度和应用程序特征相关的.

本文提出的IFBHP算法相比其他算法减少了缓存失效次数,提高了IPC.这是因为通过分支预测和访存模式学习等方法,本文算法可以准确地预测未来访存地址序列.使用更准确的分支预测算法以及支持更多的访存模式可以进一步提升IFBHP算法的性能.

6 总结

本文提出了一种基于分支预测技术和混合访存模式学习的缓存预取方法.使用分支预测技术推测程序指令流,识别指令流中的访存指令流,然后通过模式学

习对每条访存指令的地址进行预测.通过估算访存指令进入流水线的时刻,控制其预取请求的发送.本文使用GEM5模拟器搭建原型系统,分析了预取请求发送的时间窗口,并通过性能对比实验证明了本文算法的有效性.本文算法相比STeMS算法、ISB++算法、SANGAM算法、IPCP算法读操作缓存失效次数分别平均减少31.58%,28.85%,17.85%,11.48%;本文算法相比STeMS算法、ISB++算法、SANGAM算法、IPCP算法写操作缓存失效次数分别平均减少31.58%,28.85%,17.85%,11.48%.后续工作将研究如何进一步减少算法的开销,同时识别更多、更复杂的访存模式.

参考文献

[1] WULF W A, MCKEE S A. Hitting the memory wall[J]. ACM SIGARCH Computer Architecture News, 1995, 23(1): 20-24.

[2] WON J Y, GRATZ P, SHAKKOTTAI S, et al. Having your cake and eating it too: Energy savings without performance loss through resource sharing driven power management[C]//2015 IEEE/ACM International Symposium on Low Power Electronics and Design(ISLPED). Rome: IEEE, 2015: 255-260.

[3] BAKHSHALIPOUR M, TABAEIAGHDAEI S, LOTFI-KAMRAN P, et al. Evaluation of hardware data prefetchers on server processors[J]. ACM Computing Surveys, 2020, 52(3): 1-29.

[4] SMITH A J. Sequential program prefetching in memory hierarchies[J]. Computer, 1978, 11(12): 7-21.

[5] CHEN T F, BAER J L. Effective hardware-based data prefetching for high-performance processors[J]. IEEE Transactions on Computers, 1995, 44(5): 609-623.

[6] SOMOGYI S, WENISCH T F, AILAMAKI A, et al. Spatio-temporal memory streaming[C]//Proceedings of the 36th Annual International Symposium on Computer Architecture. New York: ACM, 2009: 69-80.

[7] WU H, NATHELLA K, SUNWOO D, et al. Efficient metadata management for irregular data prefetching[C]//2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture. Phoenix: IEEE, 2019: 1-13.

[8] JAIN A, LIN C. Linearizing irregular memory accesses for improved correlated prefetching[C]//2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). Davis: IEEE, 2013: 247-259.

[9] MICHAUD P. Best-offset hardware prefetching[C]//2016 IEEE International Symposium on High Performance Computer Architecture. Barcelona: IEEE, 2016: 469-480.

[10] PAKALAPATI S, PANDA B. Bouquet of instruction pointers: Instruction pointer classifier-based spatial hardware prefetching[C]//2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture. Valencia: IEEE, 2020: 118-131.

[11] CHAUDHURI M, DESHMUKH N. Sangam: A multi-component core cache prefetcher[EB/OL]. (2019-06-23)[2021-10-26]. https://dpc3.compas.cs.stonybrook.edu/pdfs/Sangam.pdf.

[12] KADJO D, KIM J, SHARMA P, et al. B-fetch: Branch prediction directed prefetching for chip-multiprocessors[C]//2014 47th Annual IEEE/ACM International Symposium on Microarchitecture. Cambridge: IEEE, 2014: 623-634.

[13] KIM J, PUGSLEY S H, GRATZ P V, et al. Path confidence based lookahead prefetching[C]//2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). Taipei: IEEE, 2016: 1-12.

[14] CAO P, FELTEN E W, KARLIN A R, et al. A study of integrated prefetching and caching strategies[C]//Proceedings of the 1995 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems. New York: ACM, 1995: 188-197.

[15] 王玉庆.基于指令流关联模式的缓存性能优化方法研究[D].北京:中国科学院大学, 2021.

[16] BAER J L, CHEN T F. An effective on-chip preloading scheme to reduce data access penalty[C]//Proceedings of the 1991 ACM/IEEE Conference on Supercomputing. New York: ACM, 1991: 176-186.

[17] ISHII Y, INABA M, HIRAKI K. Access map pattern matching for high performance data cache prefetch[J]. Journal of Instruction-Level Parallelism, 2011, 13(1): 1-24.

[18] SAIR S, SHERWOOD T, CALDER B. A decoupled predictor-directed stream prefetching architecture[J]. IEEE Transactions on Computers, 2003, 52(3): 260-276.

[19] PUGSLEY S H, CHISHTI Z, WILKERSON C, et al. Sandbox prefetching: Safe run-time evaluation of aggressive prefetchers[C]//2014 IEEE 20th International Symposium on High Performance Computer Architecture. Orlando: IEEE, 2014: 626-637.

[20] SHAKERINAVAET M, BAKHSHALIPOUR M, KAMRAN P L, et al. Multi-lookahead offset prefetching[EB/OL]. (2019-06-23)[2021-10-26]. https://dpc3.compas.cs.stonybrook.edu/pdfs/Multi_lookahead.pdf.

[21] BAKHSHALIPOUR M, LOTFI-KAMRAN P, SARBA-

ZI-AZAD H. Domino temporal data prefetcher[C]//2018 IEEE International Symposium on High Performance Computer Architecture. Vienna: IEEE, 2018: 131-142.

[22] WENISCH T F, SOMOGYI S, HARDAVELLAS N, et al. Temporal streaming of shared memory[J]. ACM SIGARCH Computer Architecture News, 2005, 33(2): 222-233.

[23] SHEVGOOR M, KOLADIYA S, BALASUBRAMONIAN R, et al. Efficiently prefetching complex address patterns[C]//Proceedings of the 48th International Symposium on Microarchitecture. New York: ACM, 2015: 141-152.

[24] NESBIT K J, DHODAPKAR A S, SMITH J E. AC/DC: An adaptive data cache prefetcher[C]//Proceedings. 13th International Conference on Parallel Architecture and Compilation Techniques. Antibes: IEEE, 2004: 135-145.

[25] BAKHSHALIPOUR M, SHAKERINAVA M, LOTFI-KAMRAN P, et al. Bingo spatial data prefetcher[C]//2019 IEEE International Symposium on High Performance Computer Architecture. Piscataway: IEEE, 2019: 399-411.

[26] KAMRUZZAMAN M, SWANSON S, TULLSEN D M. Inter-core prefetching for multicore processors using migrating helper threads[C]//Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems. New York: ACM, 2011: 393-404.

[27] KONDGULI S, HUANG M. R3-DLA(reduce, reuse, recycle): A more efficient approach to decoupled look-ahead architectures[C]//2019 IEEE International Symposium on High Performance Computer Architecture. Washington: IEEE, 2019: 533-544.

[28] MUTLU O, KIM H, PATT Y N. Techniques for efficient processing in runahead execution engines[J]. ACM SIGARCH Computer Architecture News, 2005, 33(2): 370-381.

[29] LEE, SMITH. Branch prediction strategies and branch target buffer design[J]. Computer, 1984, 17(1): 6-22.

[30] SEZNEC A, FRABOULET A. Effective ahead pipelining of instruction block address generation[C]//Proceedings of the 30th Annual International Symposium on Computer Architecture. New York: ACM, 2003: 241-252.

[31] gem The5 Simulator. A modular platform for computer-system architecture research[EB/OL]. (2018-08-29)[2021-02-22]. http://www.m5sim.org/Main_Page.

[32] GANESAN K, PANWAR D, JOHN L K. Generation, validation and analysis of SPEC CPU2006 simulation points based on branch, memory and TLB characteristics[C]//SPEC Benchmark Workshop. Berlin: Springer, 2009: 121-137.

作者简介

王玉庆 男,1987年出生,河南南阳人.2013年毕业于北京邮电大学计算机学院.其后在中国科学院软件研究所工作,现为博士研究生.主要研究方向为计算机体系结构、操作系统和缓存系统.

E-mail: yuqing@iscas.ac.cn

杨秋松 男.1977年出生,河北沧州人.2008年毕业于中国科学院软件研究所.其后在中国科学院软件研究所工作.博士,教授,博士生导师.主要研究方向为操作系统、软件工程和系统安全.