

基于决策树的间接访存数据预取激进度调节方法^①

薛峰^{②*} 韩晨吉^{**} 汪文祥^{***} 张福新^{*}

(* 处理器芯片全国重点实验室(中国科学院计算技术研究所) 北京 100190)

(** 中国科学院大学 北京 100049)

(*** 龙芯中科技术有限公司 北京 100190)

摘要 在通用处理器中,间接访存(indirect memory access, IMA)是一种常见的访存模式,目前已有许多针对间接访存的数据预取器。尽管这些预取器在单核系统中能够显著提升处理器性能,但针对单核系统设置的较为激进的预取配置在多核系统中未能带来同样显著的性能提升。这主要由于单核与多核系统中访存带宽需求的差异导致预取器的最佳激进度不同。现有的预取激进度调节算法大多需要手动设置调节规则,不仅效率较低,还存在调节策略不准确的问题。针对上述问题,本文提出了针对间接访存数据预取器的激进度调节算法 DTPAC(decision tree-based prefetcher aggressiveness controller),利用决策树算法自动训练出调节策略,避免了手动设置调节规则的缺点。实验结果表明,在间接访存数据预取器 Tyche 上,DTPAC 在四核系统中提升了 13.2% 的性能,超过现有预取激进度调节方法 FDP(feedback directed prefetching)和 CLIP,同时将内存流量降低了 20.0%。此外,在四核系统中,DTPAC 对另外 2 个间接访存数据预取器 Gretch 和 IMP(indirect memory prefetcher)性能分别提升 6.8% 和 5.5%。

关键词 预取激进度调节;间接访存;数据预取;决策树;通用处理器;多核系统

间接访存(indirect memory access, IMA)作为一种常见的访存模式,广泛存在于图算法、机器学习等领域中^[1-3]。间接访存由生产者 and 消费者构成,二者均为访存读指令,其中生产者表现出固定步长的访存行为,其结果用于索引消费者数组。典型的间接访存为 $A[B[i]]$,即顺序访问数组 B ,随后数组 B 的值用于索引数组 A 。由于值依赖关系的存在,消费者访存几乎不存在空间局部性,这导致传统预取器失效,大量的缓存缺失严重影响了程序性能^[4-6]。

为此,许多研究工作设计了针对间接访存的数据预取器。文献[7]提出了预取器 IMP(indirect memory prefetcher),针对 $A[B[i]]$ 这类访存模式进行预取。IMP 能够从访存流中计算出数组 A 的基址

和元素大小,从而推测随后访问数组 A 的访存地址。文献[8]设计预取器 Gretch,其在检测到访存步长变化时开始训练过程,从而解决了 IMP 无法在乱序处理器中工作的问题。文献[9]的预取器 Tyche 能在处理器前端识别并记录核心指令,通过提前执行的方式发送预取请求,提高了识别间接访存的能力。

虽然这些方法在单核系统中带来了显著的性能提升,但在多核系统中,这些预取器的性能提升并不明显。例如, Tyche 在单核系统中性能提升 17.1%,而在四核系统中性能仅提升 2.2%。这是因为这些方法主要针对单核系统,因此设置了较为激进的预取配置。然而,在多核系统中,访存带宽压力增大,

① 中国科学院计算技术研究所创新课题(E461100)资助项目。

② 男,1998年生,博士生;研究方向:计算机系统结构,CPU设计;联系人,E-mail: xuefeng16@mails.ucas.ac.cn。

(收稿日期:2025-01-14)

激进的预取配置会导致大量无效预取请求,浪费了宝贵的访存带宽资源,从而使多核系统中的性能提升不显著^[10-11]。

因此,本文需要设计一种能够根据实际负载动态调整预取激进度的机制。尽管已有研究探讨了如何动态调节预取激进度,但这些方法通常需要手动设置调节规则的参数^[10-14]。因此,每当处理器的结构发生变化时,都需要重新调整参数,效率低。此外,手动调节也容易导致调节策略不够准确。

针对上述问题,本文设计了一种针对间接访存数据预取的激进度调节算法 DTPAC (decision tree-based prefetcher aggressiveness controller)。该方法采用机器学习中决策树 (decision tree) 算法^[15]来自动训练调节策略,从而避免手动设置调节规则的弊端。具体而言,首先,在多个不同的预取激进度下运行程序,收集与访存相关的数据。随后,利用收集到的数据离线训练决策树模型。决策树的输入为访存相关的数据,输出则为预取激进度的调整方向,即增加、保持或减少。最后,将训练好的决策树集成到预取器中,使其能够在程序运行过程中,根据实时的访存数据动态调节预取的激进度。

本文的主要贡献有以下 3 个方面。

(1) 发现在多核系统中,现有的间接访存数据预取器因采用过于激进的预取策略,导致性能提升不显著。

(2) 提出了一种针对间接访存预取器的激进度调节方法 DTPAC,通过决策树算法避免了手动设置调节参数效率低下和不准确的问题。

(3) 在间接访存数据预取器 Tyche^[9]上,DTPAC 在四核系统中实现了 13.2% 的性能提升,优于已有的调节算法 FDP (feedback directed prefetching)^[10]和 CLIP^[14]。同时,DTPAC 对间接访存预取器 IMP^[7]和 Gretch^[8]同样有效。

1 背景介绍

间接访存由对生产者数组和消费者数组的访存操作构成。生产者数组按递增或递减的顺序依次访问,随后使用其结果直接或间接地索引消费者数组。最常见的间接访存模式为 $A[B[i]]$,即先顺序访问

数组 B ,随后用其值直接索引数组 A 。在这种模式下,消费者地址 $Consumer Address$ (即访问数组 A 的地址)表示为

$$Consumer Address = Base + Size \times Producer Data \quad (1)$$

式中: $Base$ 为数组 A 的起始地址; $Size$ 为数组 A 中每个元素的大小; $Producer Data$ 表示生产者数据,即数组 B 的值。由于消费者地址依赖于生产者的值,因此访问消费者地址通常不具备空间局部性。

间接访存广泛存在于图算法和机器学习领域。例如,为了高效地存储图结构和稀疏矩阵,压缩稀疏行 (compressed sparse row, CSR) 存储格式被广泛使用^[16-18]。CSR 存储格式由偏移数组、索引数组和数据数组组成。对于每个节点,偏移数组记录每个节点在索引数组中的起始位置,索引数组记录所有节点的邻居节点在数据节点中的位置,数据数组存储每个节点的属性数据。例如,如图 1 所示,节点 a 的邻居节点 b 和 c 存储在索引数组的索引 p 到 $p+2$ 之间,节点 b 的邻居节点 d 存储在索引 $p+2$ 到 $p+3$ 之间。此外,索引数组中的每一个值都指向数据数组中对应的数据。在 CSR 存储格式中,当遍历某个节点的邻居时,首先顺序访问索引数组,然后通过索引间接访问数据数组。此时,索引数组充当了生产者的角色,数据数组则是消费者。

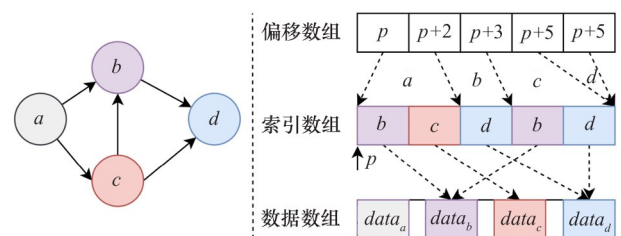


图 1 CSR 存储格式使用示例

2 相关工作

2.1 间接访存预取器

目前,许多研究工作集中于间接访存的数据预取方法。其中,部分研究基于软件或软硬件结合的预取方式。例如,ATP (array tracking prefetcher)^[4]采用软硬件结合的方法,通过编译器或程序员插入特殊指令来识别间接访存模式。APT-GET^[19]作为

一种软件预取算法,利用性能剖析生成及时的预取请求。尽管这些方法的硬件开销较小,但与纯硬件的预取器相比,这些方法需要更改源代码或编译器,从而会引发兼容性的问题。

因此,已有许多研究提出了纯硬件的数据预取器。IMP^[7]是首个针对间接访存的硬件预取算法,能够从访存地址流中识别出一些简单的间接访存模式。一旦识别成功,IMP会根据识别到的规则发送预取请求。文献[8]发现,由于乱序执行的影响,IMP难以准确识别间接访存模式,因此设计了预取器Gretch。Gretch在检测到访存步长变化时才开始识别,并采用基于窗口的机制以减轻乱序执行的影响。文献[9]发现基于访存流的识别方式效率较低,因此设计了预取器Tyche。Tyche在处理器重命名阶段记录间接访存的指令依赖链,在访存部件通过提前执行依赖链的方式发出预取请求。然而,这些方法主要讨论预取器在单核系统中的性能,并未考虑多核系统下访存带宽受限的影响。因此,这些预取器在多核系统中的性能提升并不明显,本文旨在解决这一问题。

2.2 预取激进度调节策略

为提升预取器在多核或访存带宽受限系统中的性能,诸多研究致力于动态调节预取器的激进度。在多核系统中,访存请求数量远超单核系统,导致访存带宽压力显著增加。此时,激进的预取策略可能引发大量无用预取,进一步加剧访存带宽压力,甚至对性能产生负面影响。文献[10]提出了FDP算法,该方法通过统计准确率、及时性和预取污染3个指标,动态调整预取距离。文献[11]设计了HPAC(hierarchical prefetcher aggressiveness control)调节算法,与FDP相比,HPAC额外考虑了其他处理器核对访存带宽的消耗以及各处理器核自身的访存带宽消耗情况。文献[12]针对链表数据预取器设计了激进度调节策略,但与前2个方法不同,该方案需要对编译器进行改动,由编译器指示预取器哪些指针需要预取。文献[20]针对多核系统的公平性设计了预取器激进度调节算法,采用爬山算法选择最佳预取策略。文献[13]设计了NST(near-side throttling)算法,通过检测预取的及时性,调节预取距离。

然而,目前大多数方法仍需手动设置调节规则的参数。例如在调节算法FDP中,需要定义不准确阈值,当预取器的准确率低于这个阈值时会被视为不准确。因此,当处理器微架构发生变化时,需要重新对这些参数进行手动调整。这不仅延长了开发周期,同时人工设置的阈值也可能存在不准确的情况。

3 间接访存预取器工作原理

目前尚无研究专门聚焦于调节间接访存数据预取器的激进度。间接访存数据预取器可调节的方面包括预取距离和间接访存的频率阈值。

2.1节提到的预取器IMP、Gretch和Tyche的本质类似,它们均利用Stride预取器获取的预取数据,进一步对间接访存发出预取请求。如图2所示,针对 $A[B[i]]$ 这类简单的间接访存,数据预取器的工作流程如下:图中预取距离设置为4,处理器正在访问的地址为 $B+3$,因此Stride预取器发送预取地址 $B+7$ 。当地址 $B+7$ 的数据被预取到缓存后,其值被用于计算间接访存的预取地址 $A+B^{[7]}$ 。预取距离对处理器性能十分重要,较远的预取距离能提高预取器的及时率,但过远的位置可能不会被访问,降低预取的准确率,进而导致内存带宽浪费和性能下降。

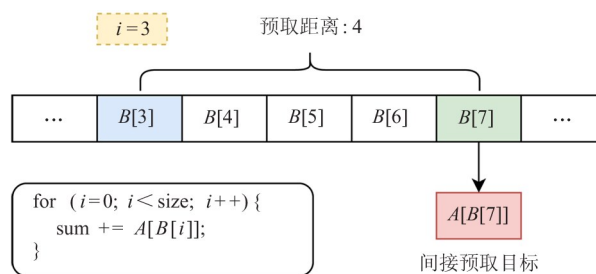


图2 间接访存预取器工作示例

此外,Tyche预取器内置了一个基于经验设置的参数——频率阈值。Tyche统计间接访存相对于固定步长访存出现的频率,只有当间接访存出现的频率超过这个阈值,间接访存才会被预取。即该参数越高,预取的条件越严格。例如,在Tyche的默认配置中,此阈值设置为50.0%,这意味着只有当间接访存出现的频率相比于固定步长访存超过50.0%

时,才会触发对间接访存的预取。研究发现,对于某些程序,在不同核数的系统中,这个参数的最佳设定也有所不同。

4 DTPAC 算法设计动机及实现

本节主要介绍 DPTAC 算法的设计动机及具体的设计实现。

4.1 设计动机

本研究在不同预取距离和不同核数系统下,测试了间接访存预取器 Tyche 的性能。实验环境和测试程序的详细描述参见第 5 节。图 3 展示了 Tyche^[9] 预取器相对于 Stride 预取器在不同配置下的性能提升结果。

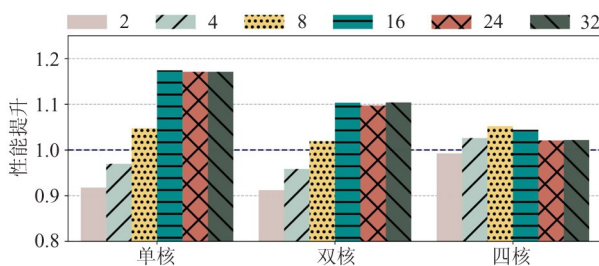


图 3 数据预取器 Tyche 在不同预取距离以及不同核数系统下相较于 Stride 预取器的性能提升

本文研究发现:(1)在预取距离设为 32 的默认配置下,预取器在多核系统中的性能提升显著低于在单核系统中的性能提升。例如,单核系统的性能提升达到了 17.1%,而四核系统的性能提升仅为 2.2%。(2)在不同核数的系统中,最佳的预取距离(即性能提升最高的预取距离)不同。在单核和双核系统中,较为激进的预取配置(预取距离大于 16)能显著提升性能。然而,在四核系统中,随着访存带宽压力的增加,激进的预取配置带来的性能提升较低。相反,在预取距离设为 8 时,性能提升效果最显著。(3)即使在相同核数的系统中,不同程序的最佳预取距离也存在差异。例如,在双核系统中, Ligr^[16] 框架的 Raddi 程序和 Triangle 程序的最优预取距离分别为 32 和 8。

因此,为了最大化间接访存预取器的性能,需要一种能够根据处理器的实时状态和应用程序的特

征,动态调节预取激进度的方法。这包括调节预取距离和频率阈值,以适应各种运行环境和应用需求。

4.2 设计实现

DTPAC 在访存系统中的位置如图 4 所示。与现有的基于间隔(interval-based)的方法相似,本方案将程序的整个执行时间划分为多个时间间隔。在每个时间间隔结束时,会根据在该间隔内收集到的数据,包括访存带宽消耗量、Stride 预取器的准确率、间接访存的出现频率,并结合当前的预取器的激进程度,决定在下一个间隔中应该增加、保持或是降低预取器的激进程度。在 DTPAC 中,时间间隔的长度设置为一百万条指令的提交。

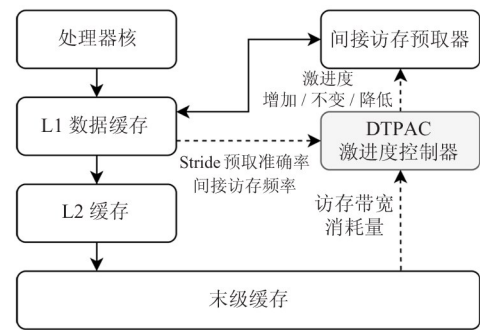


图 4 DTPAC 在访存系统中的位置

如图 5 所示,DTPAC 的整体设计流程包括 2 个离线步骤和 1 个在线运行步骤。下文将详细介绍。

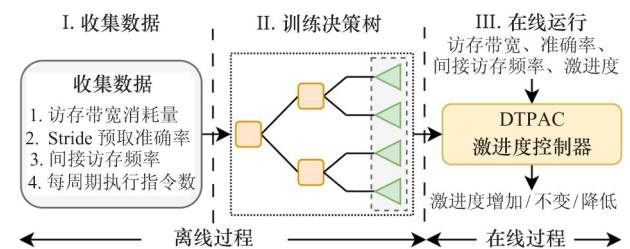


图 5 DTPAC 设计流程

4.2.1 数据收集

首先是运行应用程序,以收集后续训练决策树所需的数据。这些数据包括访存带宽消耗量、Stride 预取准确率、间接访存出现的频率以及每周周期执行指令数(instructions per cycle, IPC)。其中,访存带宽消耗量、Stride 预取准确率以及间接访存出现的频率会作为后续训练决策树的输入,IPC 用于生成训练标签。这些数据的收集是基于时间间隔的,即

对于每个程序生成多组数据,组数即为间隔数。为了训练决策树,DTPAC 在不同核数的系统中运行程序,并在每个预取激进度下收集数据。例如,在本文实现中,预取距离和频率阈值各有 6 种配置,因此预取激进度共 36 种配置。DTPAC 将在单核、双核以及四核系统中,对每个程序分别运行这 36 种激进度配置,以收集相应的数据。对于多核系统,本方案收集 0 号核的数据。

访存带宽消耗量、Stride 预取准确率以及间接访存频率是确定预取激进度的重要指标。其中,访存带宽消耗量反映了处理器的访存压力,而 Stride 预取准确率和间接访存频率则更多地反映了程序本身的特性。通过这 2 类指标的综合分析,可以确定最佳的预取策略。

访存带宽消耗量可以反映当前处理器的访存压力,从而帮助确定激进度的调节方向。例如,当访存带宽压力较高时,即使预取准确率较高,预取器可能也应当采取保守策略。因为此时应将访存带宽优先分配给真正的访存请求(demand load),而非预取请求。通过每周期末级缓存(last level cache, LLC)缺失数来衡量访存带宽消耗量。为避免浮点运算,本方案实际采用每百万周期的末级缓存缺失数。

Stride 预取准确率则能及时反映当前预取距离是否适应程序的访存行为。例如,对于访存流较短的程序,较长的预取距离会产生大量的无用预取请求,此时应降低预取距离。为测量这一指标,本文在预取器中加入了一个 32 项的先入先出(first input first output, FIFO)队列,记录所有发送的预取请求,并设有 2 个计数器分别记录有效预取请求数和预取请求总数。每发出一个预取请求,其地址会记录在 FIFO 队列中,预取请求总数加 1。每个真正的访存请求都会查找这个队列,如果命中则有效预取请求数加 1。同样地,为避免浮点运算,在计算准确率时,将有效预取请求数左移 10 位后与预取请求总数相除。因此,统计的数据是真实准确率的 1 024 倍。

另外,间接访存出现频率是评估是否应预取间接访存的关键指标。在访存带宽充足时,可以考虑预取不频繁出现的间接访存;但在访存带宽受限时,则可能需要忽略不频繁出现的间接访存。例如,

RandAcc 程序的间接访存出现频率为 50.0%。研究发现,在单核系统中对其启用间接访存预取会带来 15.6% 的性能提升,但在四核系统中反而会导致性能下降 9.3%。同样,为避免浮点运算,在统计间接访存出现的频率时,本文采用了与统计准确率相同的计算方式,即统计的数据是真实频率的 1 024 倍。

4.2.2 训练决策树

此步骤使用之前收集到的数据来训练决策树。DTPAC 将预取的激进度分为预取距离和频率阈值 2 个独立的维度,并分别对其进行调节。具体而言,预取距离设定为 6 档:2、4、8、16、24 和 32;频率阈值同样设定为 6 档:50.0%、60.0%、70.0%、80.0%、90.0% 和 100.0%。DTPAC 针对这 2 个维度分别训练决策树,两者相互独立。并在后续的在线运行中,根据当前环境独立地选择最优配置。

本文把预取激进度调节问题视为一个分类问题。同时,为了降低分类的种类,将当前激进度也作为输入,输出激进度的调节方向,即增加、保持或减少这 3 个分类。从而将 6 分类问题简化为 3 分类问题,降低决策树的复杂度。对于预取距离,决策树的输入为访存带宽消耗量、Stride 预取准确率以及当前预取距离,输出为预取距离的调整方向。对于频率阈值,输入参数包括访存带宽消耗量、间接访存出现频率以及当前阈值,输出为频率阈值的调整方向。

决策树的训练过程如图 6 所示,由 2 个步骤组成:首先,使用上一个步骤中收集到的原始数据生成训练决策树所需的训练数据;其次,利用训练数据生成决策树。

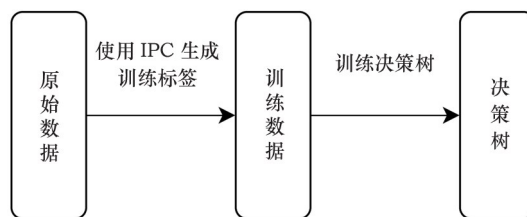


图 6 决策树训练过程

首先需要为收集到的每个时间间隔的数据生成训练标签,即预取激进度正确的调整方向。对于每个时间间隔,都有在不同的预取激进度下运行的数

据,因此可以根据不同预取激进度下的 IPC 性能确定训练标签。以预取距离的决策树训练为例,对于某个时间间隔,如果其在预取距离为 24 时 IPC 最高,那么这个间隔上预取距离为 24 的那个数据的标签设置为“保持”;这个间隔上预取距离为 2、4、8 和 16 的那些数据的标签设置为“增加”;这个间隔上预取距离为 32 的数据的标签设置为“降低”。

随后使用训练数据生成决策树模型。为验证该方法的泛化能力,本方案从收集到的数据中,随机选取 20.0% 的时间间隔数据进行训练。本文使用决策树模型^[15],并限制树的最大深度为 4,以避免决策树深度过大导致在寄存器传输级(register transfer level, RTL)实现中出现时序问题。

4.2.3 在线运行

在每个时间间隔结束时,使用访存带宽消耗量、Stride 预取准确率、间接访存出现的频率以及目前预取激进度作为激进度控制模块的输入,可得到激进度调节方向的输出,即增加、保持或减少激进度的决定,并作用于下一个时间间隔。具体而言,对于预取距离的调节,使用本间隔内收集到的程序的访存带宽消耗量、Stride 预取准确率以及当前预取距离作为输入,输出为增加、保持不变或降低预取距离。对于频率阈值,使用本间隔内收集到的程序的访存带宽消耗量、间接访存出现的频率以及当前的频率阈值作为输入,输出增加、保持或降低频率阈值。

由于决策树本质上是一系列的判断逻辑,因此可以方便地在 RTL 上实现。此外,4 层决策树结构简单,因此每个节点的判断逻辑可以通过软件配置的方式进行设置,以便软件开发人员能够对某个特定应用进行专门优化。

5 实验环境

5.1 测试程序

为验证 DTPAC 的性能表现,本文做了全面的实验评估,使用了多达 17 个测试程序,其程序算法和输入如表 1 所示。其中,Crono^[21]和 Ligma^[16]是专门针对图应用的广泛认可的基准测试套件;Graph 500 (G500)^[22]程序是对无向图执行广度优先搜索;

Hash Join (HJ)^[23]是数据库应用中常见的操作;HPCC(HPC challenge)^[24]中的 Random Access 程序随机地访问存储在大型数组中的数据;SpMV^[25]是一个用于稀疏矩阵向量乘法的基准测试。如表 2 所示,本文为这些应用程序选取了真实的图数据输入,包括从 SNAP(Stanford network analysis project)^[26]中的社交网络图结构和 SuiteSparse Matrix Collection^[27]中的流行病的马尔科夫模型。所有测试程序均基于龙芯指令集架构 LoongArch^[28]进行编译,使用的编译器为 GCC(GNU compiler collection) 8.3.0,并采用了-O3编译选项。

表 1 测试程序

测试程序	算法	输入
Crono	Community (CD), Connected Components (CC), Single Source Shortest Path (SSSP), Triangle Counting (TC), DFS	Higgs
Graph 500	BFS	s16 e10
Ligma	Bellman-Ford Shortest Paths (BF), Triangle Counting (TC), BFS, BFS-Bitvec (BFSBV), Collaborative Filtering (CF), Components Shortcut (CS), Maximal Independent Set (MIS), PageRank (PR), Radii Estimation (Radii)	Pokec
SpMV	稀疏矩阵向量乘法	mc2depi
Hash Join	哈希连接	1.28×10^8
HPCC	Random Access (RA)	64 MB

表 2 测试程序输入

输入	描述	定点数	边数
Higgs ^[26]	Twitter 上“Higgs”话题关注者网络。	456.0×10^3	14.8×10^6
Pokec ^[26]	Pokec 社交网站中用户关系。	1.6×10^6	30.0×10^6
mc2depi ^[27]	流行病的马尔科夫模型,包含 525.0×10^3 行, 525.0×10^3 列以及 2.1×10^6 非零元素。		

为加快性能评估速度,本文使用 SimPoint^[29]方法来提取程序的代表仿真点。具体来说,SimPoint 的参数设置为 $MaxK = 30, N = 100 \times 10^6$ 。对于每个仿真点,模拟器首先预热运行一千万条指令,随后再

运行一亿条指令来报告程序的性能及其他各项指标。

5.2 测试平台

本文的实验平台采用 ChampSim^[30] 模拟器。ChampSim 模拟器是一款周期级精确的 CPU 模拟器,能够准确模拟分支预测、缓存层次结构以及数据预取等微架构特性。该模拟器在学术界被广泛应用,同时也被用于第 2 届和第 3 届数据预取锦标赛。模拟器的配置信息详见表 3。

表 3 模拟器配置信息

部件	配置
Core	4 核心,4 发射及提交,224 项重排序缓存,64 项 load 队列,48 项 store 队列,TAGE-SC-L ^[31] 分支预测器
TLB (translation lookaside buffer)	64 项指令 TLB,64 项数据 TLB,1 536 项共享的二级 TLB
L1icache	私有,32 KB,8 路组相连,4 拍延迟
L1 dcache	私有,48 KB,12 路组相连,4 拍延迟
L2 cache	私有,512 KB,8 路组相连,10 拍延迟
LLC	共享,每个核 4 MB,16 路组相连,20 拍延迟
DRAM	2 通道,每通道 2 Rank,每 Rank 8 个 bank,4800 mega transfers per second

为了评估 DTPAC 的调节能力,将其与 2 个已有的研究工作进行了对比,包括预取激进度调节算法 FDP^[10] 和预取过滤技术 CLIP^[14]。目前,龙芯主流的桌面处理器包含 4 个物理核心。因此,本文主要针对四核系统这一典型环境进行测试,同时也包含单核和双核系统中的测试结果。

FDP 和 CLIP 均旨在优化内存带宽受限情况下

的预取性能。具体而言,FDP 统计了预取器的准确率、及时性和造成的污染这 3 个指标。这些指标根据预设的阈值被分级:准确率分为高、中、低,及时率分为及时和不及时,污染情况分为污染和不污染。根据这些分级结果确定预取距离的调节方向。CLIP 是针对众核系统的预取过滤技术,它统计每个访存指令和访存地址阻塞重排序缓存(reorder buffer,ROB)的次数。只有当某个指令和地址的阻塞次数超过了设定阈值,才会对该指令和地址发出预取请求,从而降低预取器对访存带宽的压力。本实验对 FDP 和 CLIP 中的阈值进行了精细的调整,以适配本文的实验环境。同时,还将这些技术与静态最优预取配置(记为 StaticOPT)进行了比较:在所有预取配置下运行程序一次,选择性能最高的配置作为 StaticOPT。

此外,为了评估 DTPAC 对不同间接访存预取器的适配能力,本文实现了 3 种间接访存预取器:Tyche^[9]、Gretch^[8] 和 IMP^[7]。这 3 种预取器的预取请求均填回至 L1 dcache,默认预取距离为 32,并且 Tyche 的默认频率阈值为 50.0%。与 Tyche 的原始实现相同,所有这些方法均基于可在多级缓存系统中协同工作的 IPCP(instruction pointer classifier based spatial prefetching)^[31] 中的 Stride 预取器。

6 实验结果与分析

6.1 性能对比与分析

图 7 展示了在四核系统中,针对间接访存预取器 Tyche 的预取调节算法相对于 Stride 预取器的 IPC

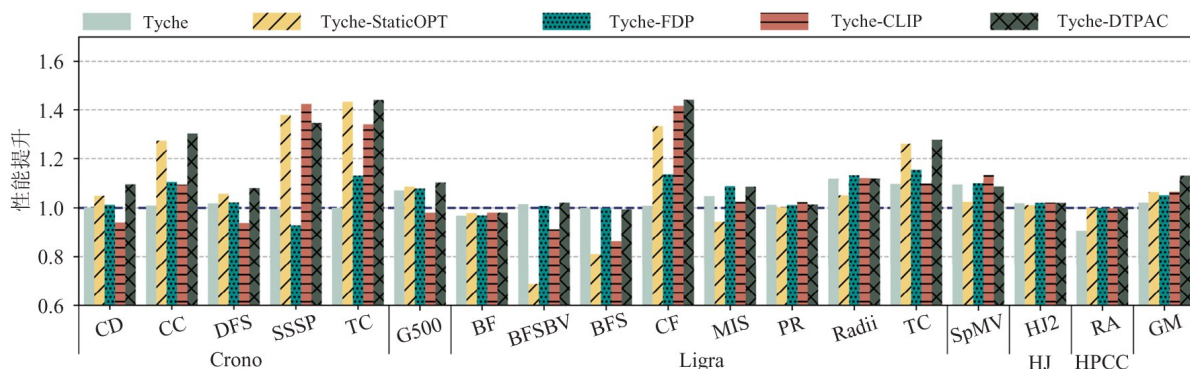


图 7 在四核系统中的性能提升

性能提升结果。IPC提升能直接反映预取器的实际效果。其中,静态最优的预取配置 StaticOPT 设置为预取距离 8 和频率阈值 80.0%。

结果显示,在四核系统中,相比于其他预取调节方法,DTPAC 取得了最高的性能提升。具体而言,Tyche、StaticOPT、FDP、CLIP 和 DTPAC 的性能提升分别为 2.2%、6.4%、5.1%、6.6% 和 13.2%。首先,DTPAC 的表现优于静态最优的预取配置 StaticOPT,这是因为 StaticOPT 虽然代表全局最优的预取配置,但是不同程序的最优配置存在差异,而 DTPAC 能够根据程序的特征及处理器实时的状态动态调整预取激进度,例如在 Crono-CD 和 Ligra-CF 上,DTPAC 性能优于 StaticOPT。另外,DTPAC 的性能提升也高于 FDP 和 CLIP,这是因为后两者需要手动设置阈值和调节规则,因此可能存在不合理的调节决策。例如,在测试程序 Crono-DFS 和 Ligra-TC 上,DTPAC 并没有作出合理的决策,以降低预取激进度。

总之,DTPAC 在多核系统中实现了显著的性能提升,这归功于其更精准的预取激进度调节能力,提升了预取器的准确率,进而降低了内存带宽的消耗。关于实验的具体数据及分析,参见下文。

6.2 覆盖率、准确率与及时率

覆盖率、准确率和及时率是评估预取器的 3 个

关键指标。本研究通过以下方式统计这些指标数据:以开启预取器后缓存未命中请求数的降低表示覆盖率;以预取器发送的有用预取请求占比表示准确率;以在实际访存请求到来前将预取数据取回缓存的预取请求占比表示及时率。其中,覆盖率展示了预取器将多少缓存缺失转化为缓存命中;准确率则显示了预取器发送的预取请求中有多少是实际上会被使用的有效请求;及时率则体现了预取器是否能够在访存请求到来之前及时将数据回填到缓存中。

如图 8 所示,DTPAC 实现了最高的准确率 91.3%,高于 FDP 的 87.1% 和 CLIP 的 89.6%,并且与 StaticOPT 的 91.3% 持平。这得益于 DTPAC 可以更好地调节预取激进度,在访存压力较大的多核系统中,能有效地降低预取激进度,从而取得了较高的预取准确率。

另外,DTPAC 的覆盖率 41.7% 和及时率 76.2% 略低于其他方法,这主要因为 DTPAC 能够有效地降低预取距离。具体而言,DTPAC 的预取距离较近,导致预取请求发送的时机较晚,导致更多的预取请求在未命中状态保持寄存器 (miss-status handling registers,MSHR) 中命中,从而降低了覆盖率和及时率。然而,对于间接访存这类内存并行度较高的访存模式,访存延迟对性能的影响并不显著。这类访存模式更关注于访存带宽,因为处理器中的 ROB 能

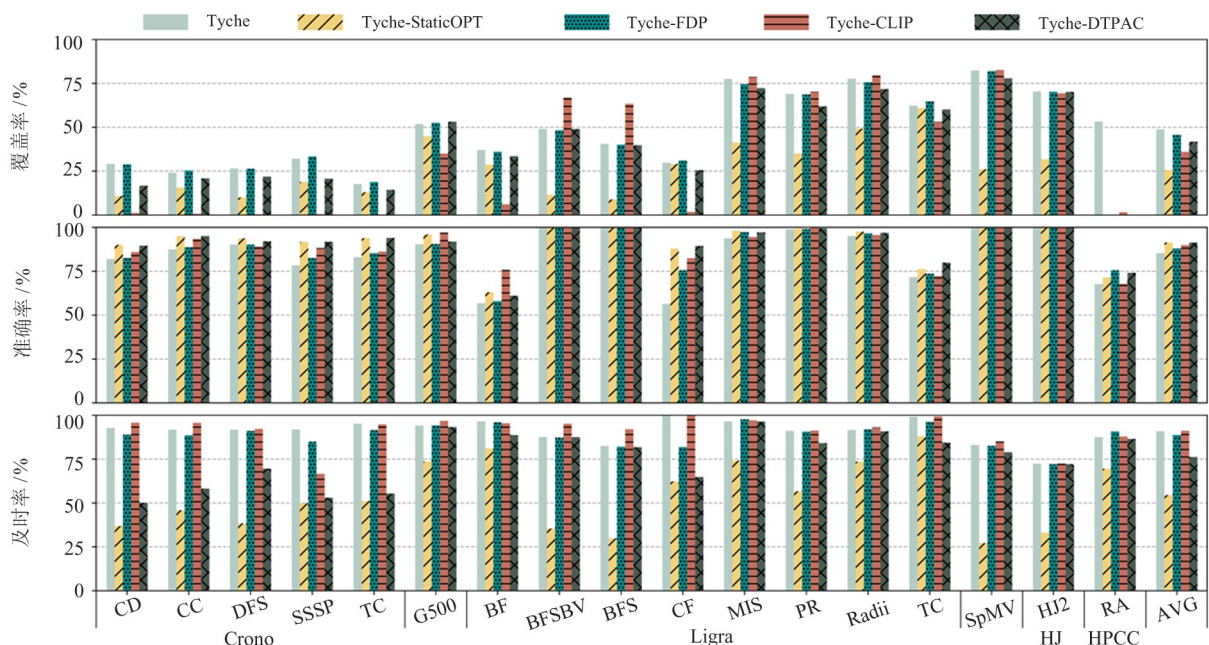


图 8 在四核系统中的覆盖率、准确率和及时率

够保证有足够多的访存请求并行执行。因此覆盖率和及时率的下降并不妨碍 DTPAC 带来显著的性能提升。此外, DTPAC 在训练决策树时, 是以 IPC 为训练目标, 因此其设计重点在于如何取得更多的性能提升, 而非更高的覆盖率和及时率。

6.3 内存带宽消耗量

图 9 展示了四核系统中, 在 Tyche 上实现不同预取算法后, 内存请求数量相比于 Stride 预取器的变化。结果表明, DTPAC 能有效降低 20.0% 的内存流量, 显著优于 StaticOPT 的 10.6% 和 FDP 的 5.5%。这是由于 DTPAC 能够有效降低预取激进度, 产生更少的无用的预取请求。例如, 在 Ligra 的 TC 程序

中, DTPAC 的准确率相较于 Tyche 提高了 11.5%, 内存带宽降低了 34.3%。尽管准确率的提升幅度较小, 但由于准确率是基于 L1 dcache 统计的, 而 L1 dcache 的访存次数远多于内存访问次数, 因此即使是较小的准确率提升也能带来显著的带宽降低。虽然 CLIP 也能将内存流量降低 25.0%, 但作为众核系统设计的预取过滤策略, 其过滤条件过于严格, 导致过滤掉了许多有用的预取请求。因此, CLIP 不适用于四核系统, 无法根据四核系统中的处理器状态进行有效调整。所以, 尽管 CLIP 在内存流量上的降低超过了 DTPAC, 但其性能提升却低于 DTPAC 在复杂应用场景中的实际价值。

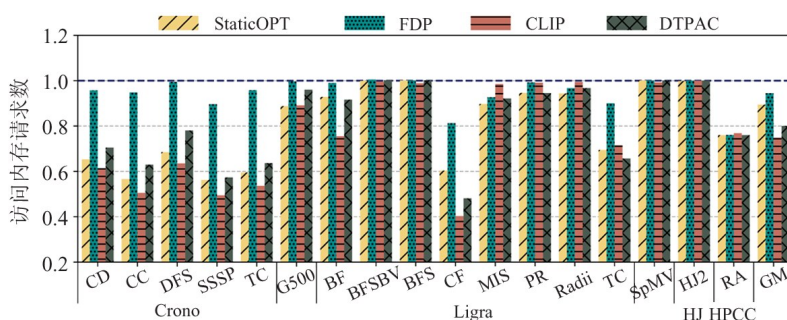


图 9 四核系统中相比于 Stride 预取器访问内存请求数变化

6.4 调节预取距离和频率阈值各自贡献

DTPAC 的调节分为 2 个相对独立的部分: 预取距离和频率阈值。为了更好地分析 DTPAC 的性能提升, 我们对这 2 个部分分别进行测试。表 4 展示了单独调节预取距离、单独调节频率阈值以及同时调节二者, 相较于预取器 Tyche 的性能提升情况。结果表明, DTPAC 的性能提升主要来自于对预取距离的调节。其中, 调节预取距离和频率阈值分别带来 9.8% 和 0.8% 的性能提升。频率阈值的调节效果不显著, 原因在于大部分程序中的间接访存是密集出现的。然而, 有 2 个程序性能提升较为明显, 分别

是 HPCC-RA 提升了 10.3%, Ligra-MIS 提升了 4.2%。因此, 对频率阈值的调节也是必要的。

6.5 单核和双核系统中性能

本文同样在单核和双核系统中测试不同调节算法的性能提升。在单核系统中, StaticOPT 的配置为预取距离 16 和频率阈值 50.0%; 在双核系统中, StaticOPT 的配置为预取距离 16 和频率阈值 60.0%。如图 10 展示了预取调节算法相对于 Stride 预取器的性能提升。结果表明, DTPAC 在单核系统中相较于默认配置性能略微提升, 达到 0.8%。这是因为在单核系统中, 内存带宽相对充裕, 预取调节机制的作用有限。在双核系统中, DTPAC 的性能提升达 3.5%, 高于 StaticOPT 的 0.1% 和 FDP 的 1.0%。此外, CLIP 的在单核和双核系统中几乎没有性能提升, 这是因为 CLIP 的过滤条件过于严格, 导致绝大部分预取请求被过滤掉, 从而使预取器的性能提升几乎为 0。

表 4 调节预取距离和频率阈值各自贡献

调节内容	整体性能提升	最大性能提升
预取距离	9.8%	43.6% (Crono-TC)
频率阈值	0.8%	10.3% (HPCC-RA)
全部	10.8%	43.8% (Crono-TC)

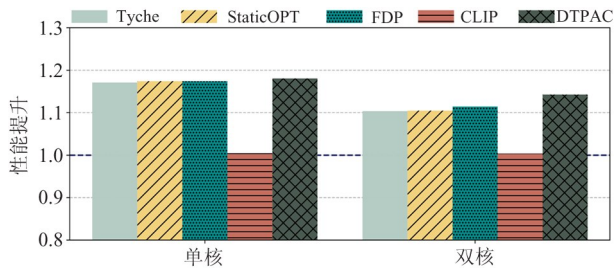


图 10 在单核和双核系统中的性能提升

6.6 对 Gretch 和 IMP 的性能提升

本文还将 DTPAC 方法应用于另外 2 个间接访存预取器 Gretch 和 IMP 上,其性能提升结果如图 11 所示。由于 Gretch 和 IMP 没有频率阈值这个设计,因此本文仅对预取距离进行调整。其中,Gretch 在单核、双核和四核系统中 StaticOPT 的预取距离分别为 16、16 和 8;IMP 在单核、双核和四核系统中 StaticOPT 的预取距离分别为 16、16 和 8。

如图 10 所示,DTPAC 在 Gretch 和 IMP 上同样表现出显著的性能提升。在四核系统中,Gretch-DTPAC 相比于 Gretch 性能提升了 6.8%,IMP-DTPAC 相比于 IMP 性能提升了 5.5%,均高于其原始设计和静态最优配置。同时,DTPAC 在单核和双核系统中性能提升也效果明显。

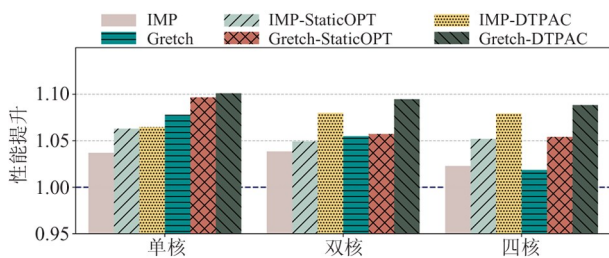


图 11 DTPAC 对 Gretch 和 IMP 的性能提升

7 结论

本文探究了现有间接访存预取器在多核系统中性能提升的局限性,并观察到在不同核数的系统中,最佳的预取激进度并不相同。为此,提出了一种基于决策树的预取激进度调节机制 DTPAC,该机制通过利用离线统计的数据自动训练预取调节规则,有效避免了手动设置调节规则效率低下和不准确的缺点。实验结果表明,在多核系统中,DTPAC 算法相

较于现有预取策略实现了显著的性能提升,并能显著减轻内存访问的压力。同时,DTPAC 还适用于多种不同的间接访存预取器。此外,DTPAC 的设计思路对于其他访存模式的预取器同样有效,这值得未来进一步深入研究。

参考文献

- [1] Kocberber O, Grot B, Picorel J, et al. Meet the walkers: accelerating index traversals for in-memory databases[C]// Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture. New York, USA:IEEE, 2013:468 - 479.
- [2] Zhang C, Zeng Y, Shalf J, et al. RnR: a software-assisted Record-and-Replay hardware prefetcher [C] // Proceedings of the 53rd Annual IEEE/ACM International Symposium on Microarchitecture. Athens, Greece: IEEE, 2020:609 - 621.
- [3] Ainsworth S, Jones T M. Software prefetching for indirect memory accesses: a microarchitectural perspective [J]. ACM Transactions on Computer Systems, 2019,36(3): 1 - 34.
- [4] Cavus M, Sendag R, Yi J. Informed prefetching for indirect memory accesses [J]. ACM Transactions on Architecture and Code Optimization, 2020,17(1):1 - 29.
- [5] Manocha A, Aragón J, Martonosi M. Graphfire: synergizing fetch, insertion, and replacement policies for graph analytics [J]. IEEE Transactions on Computers, 2023, 72(1):291 - 304.
- [6] Talati N, May K, Behroozi A, et al. Prodigy: improving the memory latency of data-indirect irregular workloads using hardware-software co-design [C]//Proceedings of the 2021 IEEE International Symposium on High-Performance Computer Architecture. Seoul, Korea:IEEE,2021:654 - 667.
- [7] Yu X, Hughes C, Satish N, et al. IMP: indirect memory prefetcher [C] // Proceedings of the 48th International Symposium on Microarchitecture. Waikiki, USA:IEEE, 2015:178 - 190.
- [8] Kaushik A, Pekhimenko G, Patel H. Gretch: a hardware prefetcher for graph analytics [J]. ACM Transactions on Architecture and Code Optimization,2021,18(2):1 - 25.
- [9] Xue F, Han C, Li X, et al. Tyche: an efficient and general prefetcher for indirect memory accesses [J]. ACM

- Transactions on Architecture and Code Optimization, 2024,21(2):1–26.
- [10] Srinath S, Mutlu O, Kim H, et al. Feedback directed prefetching: improving the performance and bandwidth-efficiency of hardware prefetchers[C]//Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture. Scottsdale, USA: IEEE, 2007:63–74.
- [11] Ebrahimi E, Mutlu O, Lee C, et al. Coordinated control of multiple prefetchers in multi-core systems[C]//Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture. New York, USA: IEEE, 2009:316–326.
- [12] Ebrahimi E, Mutlu O, Patt Y. Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems[C]//Proceedings of the IEEE 15th International Symposium on High Performance Computer Architecture. Raleigh, USA: IEEE, 2009:7–17.
- [13] Heirman W, Bois K, Vandriessche Y, et al. Near-side prefetch throttling: adaptive prefetching for high-performance many-core processors[C]//Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques. Limassol, Cyprus: IEEE, 2018:1–11.
- [14] Panda B. CLIP: load criticality based data prefetching for bandwidth-constrained many-core systems[C]//Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture. Toronto, Canada: IEEE, 2023:714–727.
- [15] Chen T, Guestrin C. XGBoost: a scalable tree boosting system[C]//Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. San Francisco, USA: ACM, 2016:785–794.
- [16] Shun J, Blelloch G. Ligra: a lightweight graph processing framework for shared memory[C]//Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. Shenzhen, China: ACM, 2013:135–146.
- [17] Gonzalez J, Xin R, Dave A, et al. GraphX: graph processing in a distributed dataflow framework[C]//Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation. Broomfield, USA: USENIX Associations, 2014:599–613.
- [18] Zhao J, Yang Y, Zhang Y, et al. TDGraph: a topology-driven accelerator for high-performance streaming graph processing[C]//Proceedings of the 49th Annual International Symposium on Computer Architecture. New York, USA: IEEE, 2022:116–129.
- [19] Jamilan S, Khan T, Ayers G, et al. APT-GET: profile-guided timely software prefetching[C]//Proceedings of the 17th European Conference on Computer Systems. Rennes, France: ACM, 2022:747–764.
- [20] Panda B. SPAC: a synergistic prefetcher aggressiveness controller for multi-core systems[J]. IEEE Transactions on Computers, 2016,65(12):3740–3753.
- [21] Ahmad M, Hijaz F, Shi Q, et al. CRONO: a benchmark suite for multithreaded graph algorithms executing on futuristic multicores[C]//Proceedings of the 2015 IEEE International Symposium on Workload Characterization. Atlanta, USA: IEEE, 2015:44–55.
- [22] Murphy R, Wheeler K, Barrett B, et al. Introducing the Graph 500[J]. Cray Users Group, 2010,19:45–74.
- [23] Balkesen C, Teubner J, Alonso G, et al. Main-memory hash joins on multi-core CPUs: tuning to the underlying hardware[C]//Proceedings of the 29th IEEE International Conference on Data Engineering. Brisbane, Australia: IEEE, 2013:362–373.
- [24] Luszczek P, Bailey D, Dongarra J, et al. The HPC challenge (HPCC) benchmark suite[C]//Proceedings of the 2006 ACM/IEEE Conference on Supercomputing. Tampa, USA: IEEE, 2006:1–11.
- [25] Liu W, Vinter B. CSR5: an efficient storage format for cross-platform sparse matrix-vector multiplication[C]//Proceedings of the 29th ACM International Conference on Supercomputing. Newport Beach, USA: ACM, 2015:339–350.
- [26] Leskovec J, Krevl A. SNAP datasets: Stanford large network dataset collection[EB/OL].[2025-01-14]. <http://snap.stanford.edu/data>.
- [27] Davis T, Hu Y. The university of Florida sparse matrix collection[J]. ACM Transactions on Mathematical Software, 2011,38(1):1–25.
- [28] 胡伟武, 汪文祥, 吴瑞阳, 等. 龙芯指令系统架构技术[J]. 计算机研究与发展, 2023,60(1):2–16.
- [29] Sherwood T, Perelman E, Hamerly G, et al. Automatically characterizing large scale program behavior[J].

- ACM SIGPLAN Notices, 2002,37(10):45–57.
- [30] Gober N, Chacon G, Wang L, et al. The championship simulator: architectural simulation for education and competition[EB/OL]. [2024–12–30]. <https://github.com/ChampSim/ChampSim>; ChampSim.
- [31] Seznec A. TAGE-SC-L branch predictors again[C]//Proceedings of the 5th JILP Workshop on Computer Architecture Competitions: Championship Branch Prediction. Seoul, Korea: JILP, 2016:1–4.
- [32] Pakalapati S, Panda B. Bouquet of instruction pointers: instruction pointer classifier-based spatial hardware prefetching[C]//Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture. Valencia, Spain: IEEE, 2020:118–131.

Decision tree based aggressiveness controller for indirect memory access prefetching

Xue Feng^{**}, Han Chenji^{**}, Wang Wenxiang^{**}, Zhang Fuxin^{*}

(^{*} State Key Laboratory of Processors (Institute of Computing Technology, Chinese Academy of Sciences), Beijing 100190)

(^{**} University of Chinese Academy of Sciences, Beijing 100049)

(^{***} Loongson Technology Co. Ltd., Beijing 100190)

Abstract

Indirect memory access (IMA) is a prevalent memory access pattern in general-purpose processors, and numerous prefetchers dedicated to handling indirect memory accesses have been developed. Although these prefetchers significantly improve the performance of processors in single-core systems, we have observed that the aggressive prefetching configurations used in single-core systems do not yield the same level of performance improvement in multi-core systems. This discrepancy is primarily attributed to the different memory bandwidth demands between single-core and multi-core systems, which result in varying optimal levels of prefetching aggressiveness. The existing methods for adjusting prefetching aggressiveness mostly involve manual setting of adjustment rules, which is not only inefficient but also prone to inaccuracies in adjustment strategies. To address these issues, this paper introduces the DTPAC (decision tree-based prefetcher aggressiveness controller), a method for automatically adjusting the aggressiveness of prefetchers for indirect memory accesses using decision tree algorithms, thereby avoiding the drawbacks of manual rule-setting. Experimental results demonstrate that on the indirect memory access prefetcher Tyche, DTPAC improves performance by 13.2% in a quad-core system, outperforming existing methods such as FDP and CLIP. Additionally, DTPAC reduces memory traffic by 20.0%. Furthermore, in a quad-core system, DTPAC also enhances the performance of two other indirect memory access prefetchers, Gretch and IMP, by 6.8% and 5.5%, respectively.

Key words: prefetching aggressiveness adjustment, indirect memory access, data prefetching, decision tree, general-purpose processor, multi-core system