



北京大学

# 硕士研究生学位论文

题目： 一种数据预取控制器的  
优化设计与实现

姓 名： 李硕轲

学 号： 1701213965

院 系： 信息科学技术学院

专 业： 计算机系统结构

研究方向： 软硬件协同设计

导师姓名： 刘先华 副教授

二〇二〇 年 六 月



# 版权声明

任何收存和保管本论文各种版本的单位和个人，未经本论文作者同意，不得将本论文转借他人，亦不得随意复制、抄录、拍照或以任何方式传播。否则，引起有碍作者著作权之问题，将可能承担法律责任。



## 摘要

随着处理器和存储器之间性能差距的不断增长，存储器性能的提升对整个计算机速度的提升起到了越来越重要的作用。数据预取是其中较为关键且已经被广泛应用的技术。但主流商用处理器中的预取器设计场景多为单核，并不能有效处理预取在多核心中的干扰等问题。同时提升预取覆盖率和提升准确率常常是相互矛盾的，预取器的性能很难得到进一步提升。本文在基于感知器设计的预取过滤器的基础上，设计并实现了多核场景下的数据预取控制器，让数据预取的性能得到了进一步的提升。本文的主要工作包括：

1. 调研了数据预取器的发展情况，并分析了经典预取器中的设计思路，对它们的优缺点进行了对比和分析。对不同预取器设计中，预取负面效应控制的方法进行了归类，调研和分析了当前比较常见的预取控制器设计思路。同时整理和对比了当下一些主流商用处理器的预取器设计。
2. 设计并实现了面向多核处理器的数据预取控制器。数据预取控制器主要包括了预取统计分析模块、预取有害性统计模块和预取过滤器模块。预取统计分析模块无需硬件上的设计实现，主要用于对预取行为、有害性的分析和统计，支持时间维度采样分析和统计分析。预取有害性统计模块可以辅助预取过滤器模块进行有害预取的训练。预取过滤器模块是在基于感知器的预取过滤器基础上扩展和优化的，本文添加了面向多核场景下的动态训练优化，扩展了训练场景和预取目标范围。在无效化、预取合并等关键操作上，保证了时序准确性。借助存储优化和自定义配置，在较低的存储开销下实现更多的功能和优化。
3. 对数据预取控制器进行了性能评测与分析。基于统计结构对预取器多核心干扰的情况进行了研究与分析，确定了多核心预取干扰的主要原因。使用 Gem5 结构级模拟器和 SPEC2006 评测程序，对实现进行了性能评测。最终本文实现的数据预取控制器在单核场景下，性能平均提升了 5.60%，其中访存密集型程序性能平均提升了 22.76%；16 核场景下访存密集型程序的性能平均提升了 0.43%。

**关键词：**数据预取，感知器，预取控制器，多核处理器，超标量处理器



# Optimal Design and Implementation of a Data Prefetch Controller

Li Shuke (Computer Architecture)

Directed by Associate Professor Liu Xianhua

## ABSTRACT

As the performance gap between processor and memory continues to increase, memory performance is playing an increasingly important role in increasing the overall computer speed. Data prefetching is one of the most important and widely used technologies. But the prefetcher in commercial processors are mostly designed for single-core scenarios, and cannot effectively deal with the problems such as the interference among prefetches from multiple cores. Improving the prefetch coverage and improving prefetch accuracy are often at odds with each other, and therefore it is difficult to further improve prefetcher's performance. Based on the perceptron-designed prefetch filter, this paper designs and implements a data prefetch controller for multi-core scenarios, which further improves the performance of data prefetch. The main work of this article includes:

1. Investigating the development of data prefetchers, analyzing the design ideas in the classic prefetcher, comparing and analyzing their advantages and disadvantages. In different prefetcher designs, the methods of prefetch negative effect control are categorized, and the state-of-the-art prefetch controller design are also investigated and analyzed. At the same time, the prefetcher designs of some mainstream commercial processors are listed and compared.
2. Designing and implementing a data prefetch controller for multi-core processor. The data prefetch controller mainly includes a prefetch statistical analysis module, a prefetch harmfulness statistics module and a prefetch filter module. The prefetch statistical analysis module does not need hardware implementation, and is mainly used for analysis and statistics of prefetch behavior and harmfulness. It also supports sampling analysis in time dimension and statistical analysis. The prefetch harmfulness statistics module will assist the prefetch filter module on training harmful prefetches. The prefetch filter module is extended and optimized on the basis of the perceptron-based prefetch filter. This paper also adds dynamic training optimization for multi-core scenarios, expands the training scenes and prefetch target range. In the key operations such as invalidation and prefetch

merge, the timing accuracy can be guaranteed. With storage optimization and custom configuration, most optimizations can be achieved with lower storage overhead.

3. Evaluating and analysing the performance of the data prefetch controller. Based on the statistical structure, the prefetcher's multi-core interference is studied and analyzed. And the main cause of multi-core prefetch interference is determined. With Gem5, an architecture-level simulator and SPEC2006 benchmark suites, performance of the implementation is evaluated. The evaluation result shows that the performance of the data prefetch controller implemented increases by an average of 5.60% in single-core scenario, among which the performance of memory-intensive programs increases by an average of 22.76%. While the performance of memory-intensive programs in 16-core scenario increases by an average of 0.43 %.

**KEY WORDS:** Data Prefetching, Perceptron, Prefetch Controller, Multi-core Processor, Superscalar Processor



# 目录

<b>第一章 绪论</b>	<b>1</b>
1.1 存储墙与存储器优化	1
1.2 单核场景下的预取与预取控制	2
1.3 主流处理器中的数据预取策略	4
1.4 多核场景下的预取优化	5
1.5 PPF 设计中存在的问题	7
1.6 论文组织结构	7
<b>第二章 数据预取负面效应控制技术</b>	<b>9</b>
2.1 数据预取	9
2.2 数据预取负面效应控制	10
2.2.1 数据预取缓冲区	10
2.2.2 基于有效信息调控数据预取	13
2.2.3 针对预取优化的缓存替换策略	14
2.2.4 缓解多核心的预取干扰	15
2.3 数据预取负面效应控制技术对比	16
2.4 本章小结	18
<b>第三章 数据预取控制器的设计与实现</b>	<b>19</b>
3.1 Gem5 的 Cache 设计分析	19
3.1.1 Gem5 的 Cache 结构设计	19
3.1.2 Gem5 的请求传输设计	20
3.1.3 Gem5 的数据预取设计	22
3.1.4 Gem5 中基于步长的预取器	23
3.2 数据预取控制器设计概述	25
3.2.1 基本概念	25
3.2.2 整体结构设计	26
3.3 数据预取统计分析模块	27
3.3.1 数据预取的分类	27
3.3.2 预取统计分析模块的设计与实现	28
3.3.3 预取统计分析模块的统计变量及更新策略	33
3.4 数据预取有害性统计模块	35

3.4.1 模块结构设计 .....	36
3.4.2 有害性的记录与更新 .....	37
3.5 数据预取过滤器模块 .....	38
3.5.1 模块结构设计 .....	39
3.5.2 预取目标层级的变更 .....	42
3.5.3 预取权重训练 .....	47
3.5.4 预取过滤 .....	50
3.6 数据预取控制器的存储开销 .....	51
3.7 本章小结 .....	53
<b>第四章 数据预取控制器的评测与分析 .....</b>	<b>55</b>
4.1 评测环境 .....	55
4.1.1 评测平台的校正 .....	55
4.1.2 评测程序 .....	57
4.1.3 评测硬件环境 .....	59
4.2 性能评估标准 .....	59
4.3 多核预取干扰分析 .....	59
4.4 数据预取控制器的性能分析 .....	62
4.4.1 单核场景性能评测 .....	63
4.4.2 多核场景性能评测 .....	70
4.5 配置对数据预取控制器性能的影响 .....	71
4.6 本章小结 .....	73
<b>第五章 总结与展望 .....</b>	<b>75</b>
5.1 本文主要工作 .....	75
5.2 未来工作展望 .....	76
<b>参考文献 .....</b>	<b>77</b>
<b>致谢 .....</b>	<b>81</b>

## 图目录

图 1.1 处理器和存储延迟变化趋势图 .....	1
图 1.2 SPP 预取准确度与预取深度的关系 .....	4
图 1.3 SPEC2006 中的多核心预取干扰情况 .....	6
图 2.1 数据预取器的分类 .....	10
图 2.2 简单的顺序流预取器结构 .....	11
图 2.3 使用预取缓冲区的内存测预取器 .....	12
图 2.4 B-Fetch 数据预取器的结构 .....	13
图 2.5 考虑预取情况下的最优替换 .....	14
图 2.6 多核场景下的预取控制器设计分类 .....	15
图 2.7 HPAC 统计的带宽信息 .....	16
图 3.1 Gem5 中双核处理器的存储层次结构 .....	20
图 3.2 Gem5 中基于 Port 的连接结构 .....	21
图 3.3 Gem5 中 L2Cache 的预取器结构 .....	22
图 3.4 基于步长预取器的训练与预取过程 .....	24
图 3.5 数据预取控制器的整体结构 .....	26
图 3.6 数据预取分类统计模块结构 .....	29
图 3.7 预取在预取统计分析模块中记录状态的变化 .....	31
图 3.8 预取处理时间的组成 .....	33
图 3.9 预取有害性统计模块的设计结构 .....	36
图 3.10 数据预取过滤器的单模块结构 .....	39
图 3.11 Cache 中生成预取训练事件的结构 .....	40
图 3.12 多模块下预取过滤器的结构设计 .....	41
图 3.13 训练事件分发器的结构 .....	42
图 3.14 基于固定步长预取器中的重复预取 .....	44
图 3.15 Gem5 中 MSHR 的工作机制 .....	45
图 4.1 bzip2 中不同类型的预取个数 .....	60
图 4.2 bzip2 中预取不同阶段的处理时间 .....	61

图 4.3 不同参数对 bzip2 性能的影响 .....	62
图 4.4 单核场景下预取属性的变化 .....	63
图 4.5 单核场景下的性能提升 .....	64
图 4.6 单核场景下不同类型预取占比 .....	65
图 4.7 不同配置下数据预取控制器的性能提升 .....	68
图 4.8 L1DCache 使用数据预取器的性能变化 .....	68
图 4.9 不同 Cache 配置下的性能变化情况 .....	69
图 4.10 16 核 16 进程场景下的性能提升 .....	70
图 4.11 16 核 16 进程场景下预取干扰的变化 .....	71
图 4.12 使用不同位数 Tag 对性能的影响 .....	72
图 4.13 表格大小对性能的影响 .....	72
图 4.14 训练事件队列配置对训练的影响 .....	73

## 表目录

表 1.1 数据预取的常见类型和对应预取器 .....	3
表 1.2 主流商用处理器架构中的预取策略 .....	5
表 2.1 预取的主要分类和优缺点 .....	9
表 2.2 不同预取负面控制技术的效果对比 .....	16
表 2.3 预取负面效应控制技术的局限性 .....	17
表 3.1 基于固定步长的数据预取器默认配置 .....	23
表 3.2 多核场景下的预取分类条件 .....	28
表 3.3 预取分析统计更新流程 .....	30
表 3.4 预取统计分析模块的统计数据 .....	34
表 3.5 非共享 Cache 中预取 MSHR 的请求合并规则 .....	46
表 3.6 共享 Cache 中预取 MSHR 的请求合并规则 .....	47
表 3.7 数据预取过滤器模块使用的特征 .....	48
表 3.8 预取过滤器模块的训练事件分布 .....	49
表 3.9 预取过滤器模块的训练配置 .....	49
表 3.10 预取有害性统计模块的存储开销 .....	52
表 3.11 数据预取控制器用于训练的元数据 .....	53
表 3.12 数据预取过滤器模块的存储开销 .....	54
表 4.1 评测使用的默认 Cache 配置 .....	55
表 4.2 评测使用的 CPU 配置及校正 .....	56
表 4.3 SPEC2006 评测程序的基本信息 .....	57
表 4.4 评测程序的编译环境 .....	58
表 4.5 宿主服务器配置表 .....	58
表 4.6 数据预取控制器对性能影响的分析 .....	65



## 第一章 绪论

本章主要介绍了本文的研究背景与意义。本章首先对存储墙和主流的存储器优化技术进行了简单介绍，然后归纳了国内外学者在数据预取器和控制器方面的研究。整理了部分主流处理器中的预取器设计，并讨论了多核心场景下优化数据预取的意义和基于感知器的预取过滤机制存在的问题。最后介绍了论文的组织结构。

### 1.1 存储墙与存储器优化

目前主流的计算机都是基于冯诺伊曼结构设计的，存储器和处理器均为其中的关键部分。尽管现代处理设计已经进入了多核时代多年，但 Amdahl 定律<sup>[1][2]</sup>中提到，程序中不可并行执行的部分是突破性能提升上限的关键。因此单个处理器核心的性能依旧对系统整体性能有着重要的影响。经过多年的发展，虽然处理器和 DRAM 存储器的速度都呈现指数增长的趋势，但是存储器速度提升的指数层级却不及处理器（如图 1.1<sup>[3]</sup>所示）。更加严重的是，两者速度差异增长也是指数级别的，这使得存储墙的问题愈发严重<sup>[4]</sup>，而存储器性能的提升对整个计算机速度的提升起到了越来越重要的作用<sup>[5]</sup>。为了有效地掩盖存储访问所带来的开销，存储层次结构的思想被提了出来。但是高效的存储器所带来的硬件成本往往更高，因此需要在存储器性能和成本开销之间做好权衡。

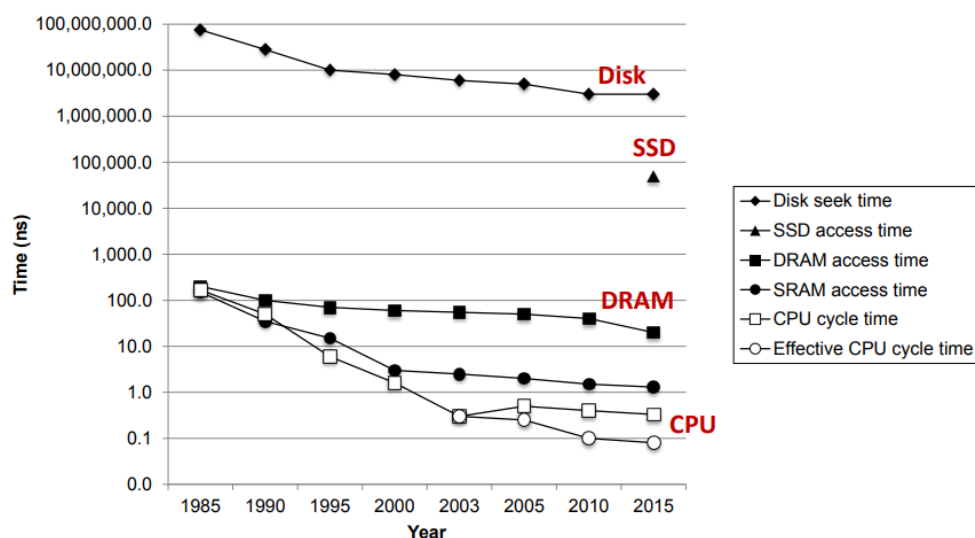


图 1.1 处理器和存储延迟变化趋势图

现代存储器层次结构中，越接近处理器的存储层次结构容量越小、访问速度越快、单字节的制造成本越高。接近处理器的 Cache 结构往往无法存放完整的工作集数据，这就使得提高 Cache 的性能成为了存储器优化工作中的核心内容<sup>[5]</sup>。而目前比较常见

的优化方案主要是使用大尺寸 Cache、多线程切换和预取技术。

使用大尺寸 Cache 虽然可以有效提升 Cache 的命中率,但是也会增加访问的命中时间,并占用宝贵的片上资源,导致处理器功耗的增加<sup>[7]</sup>。实际上,人们更倾向于将大尺寸的 Cache 设计在核心数目较多的处理器中。

使用多线程切换优化 Cache 性能,是指当处理器中的一个线程因为 Cache 访问缺失而进入等待的过程时,通过切换到其他可运行的线程继续执行,来掩盖时间开销。多线程切换的效果十分有限,并且仅仅适用于线程上下文切换时间小于 Cache 缺失所需处理时间的处理器中<sup>[5]</sup>。

相比之下,预取技术所引入的硬件成本和功耗开销都要小很多,并且拥有更好的优化效果。对于部分工作集,优秀的预取策略甚至可以避免大部分 Load 指令的 Cache 访问缺失<sup>[8][9]</sup>。因此预取技术已经被广泛地应用于当下主流的商用处理器中。

但是预取并不完美。过于激进的预取可能会向 Cache 中引入无用的数据,甚至替换掉有用的 Cache 数据<sup>[10]</sup>。同时随着多核处理器的出现,针对于单核心的传统预取技术在多核处理器中往往并不能达到理想的效果,主要是因为不同核心之间预取请求存在的干扰。本文将会针对多核处理器提出一种预取控制器的设计,来预防有害预取对正常的数据造成损害,同时进一步提升预取的性能,并对其展开性能评测与分析。

## 1.2 单核场景下的预取与预取控制

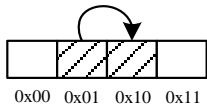
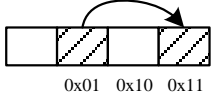
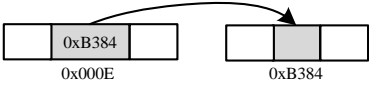
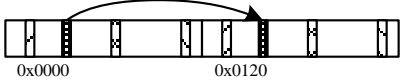
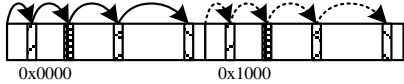
现代处理器大多会将数据和指令分开存放放到 L1DCache 和 L1ICache 中,主要是因为指令数据和非指令数据的访问习惯和访问足迹差异较大。指令访问模式很大程度上与程序的执行路径相关,并且在没有遇到分支指令时,指令访问一定是简单的连续访问,空间局部性好。对于一些商业型应用的工作集,由于代码量和指令足迹很大,L1Cache 和 L2Cache 的命中率并不高,因此指令预取对这些情景更加重要。相对的,非指令数据的访问模式和工作集相关性更强,同时其访问模式的也更加复杂,因此数据预取的难度更大,对所有应用场景都十分重要。

表 1.1 展示了应用中较为常见的计中数据访问模式,以及以相应访问模式为核心目标设计的预取器。前四种访问模式更早地被研究者挖掘出来,而近几年的研究则开始专注于基于页表访问模式的预取上。目前主流操作系统都是基于页面对内存进行管理的,因此在页表内尝试寻找固定的访问模式来进行预取的策略认可度很高。而不规则数据访问出现的场景最为广泛,处理难度也是最大的。尽管早在 1997 年 Joseph<sup>[21]</sup>就提出了使用 Markov 算法进行不规则的数据访问预取的思路,但是受限于硬件开销,它的很难被高效地实现。而诸如 BOP (Best Offset Prefetcher)<sup>[22]</sup>这样将类似 Markov 算法地预测范围限制在页表内,并结合页表访问规律进行预取的设计在近期也得到了较为广



泛的认可。

表 1.1 数据预取的常见类型和对应预取器

预取类型	出现场景	说明简图	针对的预取器
相邻缓存行访问	空间局部性强的数据访问		Next-Line Prefetcher <sup>[11]</sup> 、Adjacent Prefetcher <sup>[12]</sup> 、Streaming Prefetcher <sup>[13]</sup>
固定步长访问	循环访问数组		Stride Prefetcher <sup>[14]</sup> 、PC-based Stride Prefetcher <sup>[15]</sup>
基于指针的访问	常见 C 程序		Recursive Data Structures Prefetcher <sup>[16]</sup> 、Jump-Pointer Prefetcher <sup>[17]</sup>
Same-Object 访问	面向对象的语言常出现		Memory binding and group Prefetcher <sup>[18]</sup>
基于页表的访问	支持页表的系统执行应用时		Bingo Prefetcher <sup>[19]</sup> 、Variable Length Delta Prefetcher <sup>[20]</sup>
不规则访问	任何场景	无	Markov Prefetcher <sup>[21]</sup> 、Best Offset Prefetcher <sup>[22]</sup> 、Signature Path Prefetcher <sup>[23]</sup>

但是在数据预取器中提升预取覆盖率和提升准确率常常是相互矛盾的，盲目提升预取器激进度并不一定能够获得性能地提升。图 1.2<sup>[24]</sup>展示了 Kim 等人设计的 SPP（Signature Path Prefetcher）<sup>[23]</sup>预取器，在 SPEC2017<sup>[25]</sup>的测试子集 603.bwaves\_s 中有效预取数目随预取激进度提升的变化情况。可以发现，随着预取深度的增加，有效预取数（PF\_GOOD）的占总预取数（PF\_TOTAL）的比率越来越低，同时 IPC 也因此越来越小。可见预取器生成的非有效预取对系统的性能造成了严重的影响。在多核心系统中，非有效预取会聚集在 LLC（Last Level Cache）中，并对性能产生更大的影响。

多数数据预取器在设计时，常常会添加额外的结构自行对预取进行过滤，来避免产生不准确的预取请求。这个过程可以是基于训练可信度进行的，也可以基于预取对性能影响的反馈进行的。但由于预取器只能看到核心内的反馈信息，其预取过滤效果比较有限。因此由外部结构基于全局的预取反馈信息，控制预取来提升预取器效能，已经成为了一个独立的研究方向。

在软件方面，Intel 不仅支持获取最近 CPU 的性能参数，还允许程序员通过特定的指令关闭和开启某一个层级的预取器。Intel 处理器虽然可以通过软件实现预取器的调控，但是调控过程会占用额外的计算资源，同时仅仅依靠少量的性能参数并不能得到

准确的预取调控处理。在硬件方面, Ebrahimi<sup>[43]</sup>和 Albericio<sup>[32]</sup>则分别从 Cache 和 DRAM 层级, 设计了依据带宽占用情况调节预取器激进程度的控制器。他们对预取在不同 Bank 中产生的带宽压力进行评估, 依据评估结果对某一个核心的预取器进行调整来达到控制预取的效果。Bhatia E<sup>[24]</sup>则基于感知器的原理对预取进行细粒度的过滤, 通过预取的命中的情况训练预取不同特征的权重, 基于权重确定预取数据应该存放到当前 Cache 还是低层级的 Cache 中。由于基于感知器原理的细粒度过滤可以在保证预取准确率的同时, 进一步提升预取覆盖率, 让预取性能得到进一步的提升, 是目前最先进的预取过滤策略。因此本文选择在 Bhatia E 所设计的 PPF (Perceptron-based Prefetch Filter)<sup>[24]</sup>基础上开展预取控制器设计工作, 进行进一步的优化, 以期获得更多的性能提升。

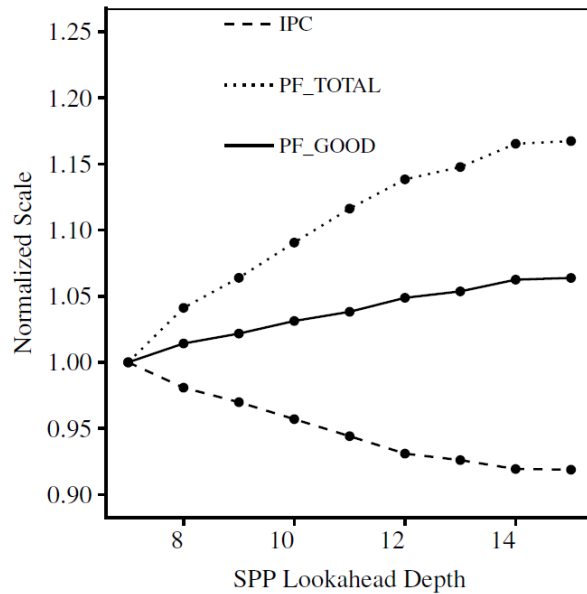


图 1.2 SPP 预取准确度与预取深度的关系

### 1.3 主流处理器中的数据预取策略

预取技术诞生于上个世纪末, 因此很早便应用到了商用处理器中。表 1.2 展示了部分主流商用处理器架构中所使用的预取器设计, 可以看到无论是在 L1 DCache 中还是在 L2Cache 中, 它们使用的依旧是十年前甚至是二十年前提出的预取策略。

Intel 的 Sandy Bridge 微架构<sup>[26]</sup>在 L1DCache 中使用的 Next-line Prefetcher, 是在 Jouppi 设计的顺序流预取器基础上改进得到的。Stride Prefetcher 则可以预取步长最多为 2KB 的固定步长访问流, 两者均不支持跨页预取。在 L2Cache 和 LLC 中, Sandy Bridge 使用了基于历史访问流的流预取器和相邻缓存行预取器。

AMD 的 15h 系列处理器<sup>[27]</sup>在 L1DCache 仅仅使用了一个基于步长的预取器, 同时支持的步长最多为 512Bytes。在 L2Cache 中, 为了兼顾不规则数据访问的情景, 除了

基于步长的预取器以外，还加入了针对不规则访问的预取器。这两个预取器共享存储结构，同一个存储表项既可以存放不规则预取器使用的的不规则访问模式，也可以存放规则访问流的步长信息。此外，15h 处理器 L2Cache 中的预取器可以使用 L1DCache 的信息进行辅助训练以提升预取准确度。

Arm 受限于硬件资源和功耗限制，无法为了提升预取的覆盖率，使用多种预取器。在 Cortex-A65 的架构<sup>[28]</sup>中，仅仅在 L1DCache 使用了 Stream Prefetcher。该预取器基于历史访问模式进行预取，允许指定预取数据存放到指定层级的 Cache 中，同时也支持软件控制和跨页预取。

表 1.2 中所列出的主流处理器虽然使用了多种预取器来提升预取的覆盖率，但是这些预取器多为单核场景下设计，依旧无法有效确保多核环境下产生的预取不会存在严重的干扰。因此如何有效处理多核心场景下的预取过滤，是一个值得研究的内容。

表 1.2 主流商用处理器架构中的预取策略

公司	微架构/ 处理器	Cache 层级	预取器	说明
Intel	Sandy Bridge	L1D	Stride Prefetcher & Next-line Prefetcher (Streaming Prefetcher)	步长多达 2KB，不支持跨页预取
		L2&LLC	Stream Prefetcher & Adjacent-line Prefetcher (Spatial Prefetcher)	不支持跨页预取
AMD	15h Processors	L1D	Stride Prefetcher	L2 辅助训练，步长多达 512Byte，不支持跨页预取
		L2	Stride Prefetcher & Irregular Prefetcher	支持 4096 个不规则访问模式或者规则访问流，不支持跨页预取
Arm	Cortex-A65	L1D	Stream Prefetcher	支持跨页预取

## 1.4 多核场景下的预取优化

随着多核处理器的普及，不同核心对共享存储层次结构带宽及存储资源的争用以及不同核心间预取数据污染的问题就一直是相关领域科研人员研究的重点。在现代主流处理器设计中，每个核心的私有 Cache 上都会设计相应的预取器。而未考虑多核场景的预取设计，无论是在存储带宽方面，还是核心间预取数据污染方面都可能会产生不良的影响。

图 1.3 展示了在 SPEC2006<sup>[29]</sup>中部分程序多进程并行执行时，基于步长的预取器在多核心场景下产生的干扰情况。测试在 16 核心，私有 L1/L2Cache 和共享 L3Cache 的结构上进行，计算方法以及更详细的实验环境配置，会在第四章评测部分进行更具体的说明。原始干扰主要来自于非预取访问时，不同核心在带宽、LLC 等共享资源的争用。从图 1.3 中可以看到随着运行进程数目的增加，多数测试程序的性能出现了下降。在 401.bzip2、445.gobmk、458.sjeng 等测试集中，预取产生的多核干扰随同时执行进程数的增加显著提升。来自多核心预取的干扰最多使性能下降了 40%以上，同样情况下，原始干扰带来的性能损失也则达到了 50%。

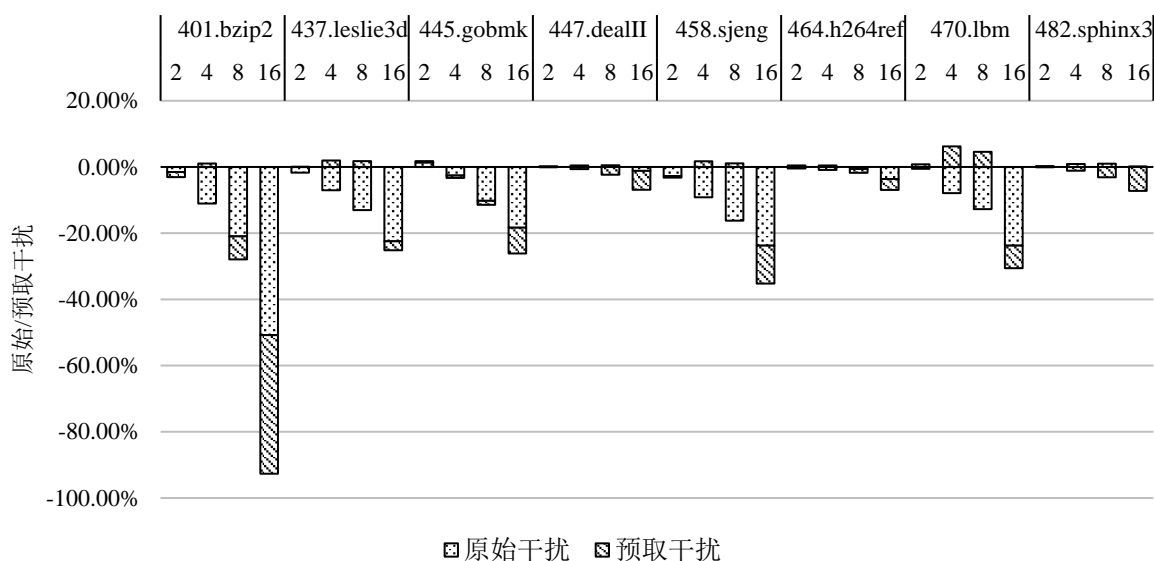


图 1.3 SPEC2006 中的多核心预取干扰情况

如果可以有效地避免不同核心之间预取的干扰，就可以使得预取的效率得到提升。而避免有害预取，则可以使预取以外的 Cache 访问性能不会受到较大的影响。两者都可以使执行 Load/Store 指令时，发生 Cache 访问缺失的比率进一步降低，从而有效地提升处理器的 IPC (Instructions Per Cycle):

- 减轻因多核心预取数据污染产生的干扰

在多核心处理器设计中，几乎都会设置共享的 L2Cache 或者共享的 L3Cache<sup>[30]</sup>。在包含型高速缓存层次结构中，不同核心执行程序的数据会出现在共享 LLC 中。执行访问密集型程序的核心预取请求更多，也常常会占用较大的 LLC 空间。在这种情况下，执行访问密集型程序核心的预取数据很可能会将其他核心正在使用的数据替换出去。这会导致其他核心私有 Cache 中存放的数据失效，产生额外的 Cache 访问缺失开销<sup>[31]</sup>。如果可以有效过滤那些会导致其他核心严重性能损失的预取请求，或者避免预取操作替换其他核心正在使用的数据，就可以有效地减轻多核心预取操作之间的干扰。

- 减轻因多核心预取带宽争用产生的干扰

即使在多核心处理器中，每一个预取请求都要经过片上网络传送到主存的访问总线上。由于预取请求会与需求请求（Demand Request）共享数据访问的 IO 通道，因此多核心处理器中预取对存储带宽产生的压力往往要比单核处理器更大。针对这种情况，往往可以通过对内存控制器的请求处理优先级，使用基于反馈的预取策略等方法来减缓带宽压力。通过有效地均衡数据访问带宽的压力，可以在一定程度上保证预取和数据访问操作得到及时的处理<sup>[32]</sup>。

## 1.5 PPF 设计中存在的问题

本文所设计的预取控制器是在 Bhatia E 提出的 PPF<sup>[24]</sup>的基础上优化和扩展得到。PPF 基于感知器的方法对预取进行有效的细粒度过滤，可以在保证预取准确性的同时，有效提升预取的覆盖率。但是 PPF 的设计中依然存在着几个问题，而这几个问题便是本文设计不同结构进行优化的核心：

- 不支持跨层级预取训练，训练场景单一：PPF 的设计只能从所在 Cache 层级获取信息进行预取的训练，如果预取数据被放置在低等级的 Cache 中，将不会进行训练。同时 PPF 支持的训练场景较为简单，错失了大量的预取训练机会，使得预取的训练效果受限。
- 未考虑对有害预取进行惩罚训练：只考虑了无用预取和有效预取相关的训练，针对有害预取没有额外的惩罚训练，使得预取训练效果欠佳。
- 没有有效的多核预取训练支持：PPF 的设计并不支持不同核心之间预取训练信息的共享，只能使用非全局信息进行预取训练，使得在单核场景下与多核场景下的训练方法没有区分度。
- 预取过滤时只能降低预取目标层级：PPF 只能降低预取数据存放的 Cache 等级，或者讲预取过滤掉，对于准确的预取，却不能将预取数据放到更高层级的 Cache 中以获得更好的预取效果。

本文则通过添加多个新的功能模块配合 PPF 进行预取训练，并结合 PPF 基础策略的扩展和优化，有效的解决了上述几个问题，最终在使用极少的额外硬件开销的情况下，在性能上获得了有效的提升。

## 1.6 论文组织结构

论文的组织结构如下：

第一章是绪论。该章节主要介绍了本文的研究背景和意义，简单分析了国内外学者对数据预取器及数据预取器的研究。并简要说明了单核场景下预取存在的问题，多核场景下优化数据预取的必要性以及基于感知器的预取过滤策略中的问题。

第二章是数据预取负面效应控制技术的研究。该章节分析了历史上比较经典的数据预取器中在预取负面效应控制方面的设计思路，并对它们进行了简单的归纳和整理。最后讨论和确定了在多核心场景下可行的预取控制器实现方案。

第三章是数据预取控制器的设计与实现。该章节首先介绍了 Gem5<sup>[33]</sup>模拟器中对数据预取的支持方式。然后分别从预取分类统计模块、预取有害性统计模块、预取过滤器模块三个方面介绍数据预取控制器的设计与实现。同时包括了对预取层级变更有效实现，有害性统计和训练策略，控制器的过滤策略等设计细节的详细介绍。

第四章是数据预取控制器的性能评测与分析。本文在结构级模拟器 Gem5 上，对 SPEC2006 中的测试集进行测试。通过改变预取器配置、Cache 配置等配置参数来观察对预取行为的影响，确定了产生多核心预取干扰的主要原因。对数据预取控制器实现进行了性能评测。主要内容包括评测环境的介绍、评测指标的确定和评测结果的分析。对比了不同情况下使用数据预取控制器的性能变化，并确定了存储开销优化对数据预取控制器性能的影响。

第六章是总结与后续工作的展望。对已完成的工作进行总结和分析，并对数据预取控制器的下一步工作进行了展望。

## 第二章 数据预取负面效应控制技术

国内外在数据预取技术方面的研究层出不穷，这些技术大多需要占用额外的硬件资源。没有哪一种预取器可以做到完全理想，不产生任何预取负面效应。在控制预取负面效应方面，研究人员也做过很多相关的工作，比如添加预取缓冲区，动态调整预取激进程度等等。这些预取负面效应控制方法的优缺点各不相同，本文则选择了其中最合适一个作为研究的核心方向。

### 2.1 数据预取

预取是针对存储结构层次主要的优化技术之一，在计算机体系结构领域有很多的学者都在致力于相关的研究中。这些研究针对于不同层次，或者不同场景下的预取进行了针对性优化。表 2.1 展示了预取研究方向的主要分类，该表分别从三个方面对预取器的设计进行了划分，而本文的研究重点在于采用硬件设计的核心侧数据预取。

表 2.1 预取的主要分类和优缺点

分类维度	分类	优点	缺点
基于实现方法	硬件预取	动态发现不同模式的访问、准确率高	占用片上硬件资源
	软件预取	不占用片上资源	基于静态分析，准确率低，工作集影响大
基于预取器数据存储位置	核心侧预取	预取器数据存放在核心附近，处理速度快	复杂的预取器会占用较多的片上硬件资源
	内存侧预取	预取器数据存放在内存中，不占用片上资源，可存放数据量大	数据处理速度较慢，需要等待从内存获取的数据生成预取请求
基于目标数据类型	指令预取	空间局部性高，与控制流密切相关	会受到分支预测准确率的影响
	数据预取	无明显优点	空间局部性受工作集影响大，访问模式复杂，预取负面效应大，处理难度大

相比于指令预取，数据预取的难度更大。指令预取的空间局部性更好，并且常常和控制流密切相关。基于分支预测器的信息，可以做到十分准确的指令预取。相比之下，数据预取的难度更大，输入集的动态变化进一步降低了数据预取的可预测性。虽然数据预取也具有空间及时间局部性，并且和控制流有关联，但是这些特征都没有指令预

取明显。加上现代编程语言所带来的复杂数据访问模式，商用应用中复杂多变的工作集，在有限硬件资源的限制下做出优秀数据预取器的难度变得越来越大。国内外学者依旧在不断地研究和发现新的数据预取设计思路，这些设计随着软件的更新而变化，种类也变得越来越多样。

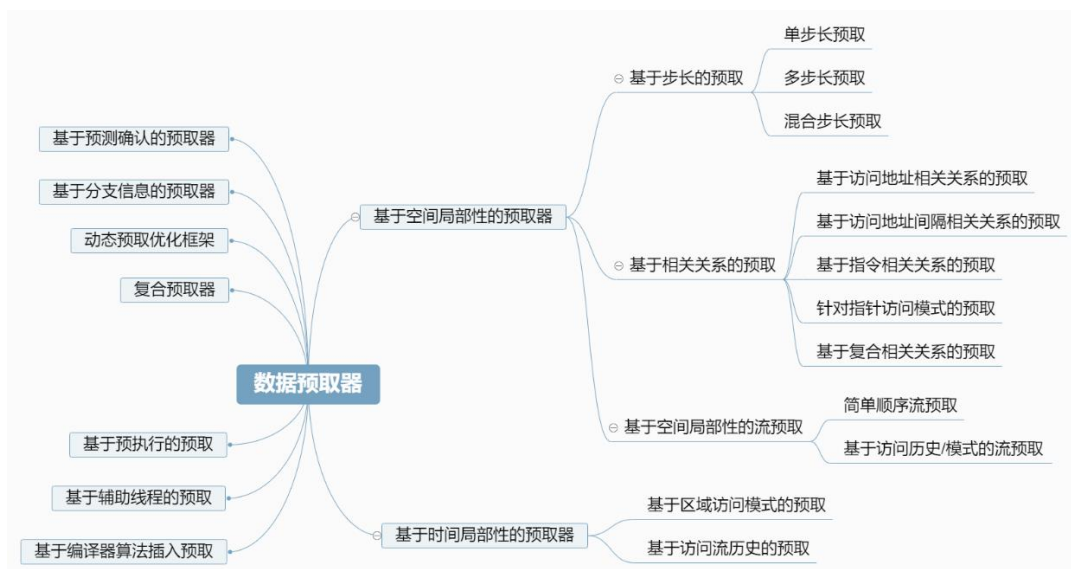


图 2.1 数据预取器的分类

图 2.1 展示了经典数据预取器的分类情况，不同数据预取器设计存在着不同的缺点。基于步长的预取只能侦测固定步长访问的场景，针对不规则预取效果比较差。如果设计使用固定的预取激进度，那么产生的不准确预取请求会占用较多的存储带宽，对性能产生损害。而基于相关关系的预取常常需要很大的存储资源存放相关关系，以获得更好的预取覆盖率。无论是哪一种预取器，都不可能做到 100% 的预取准确率，而不准确的预取常常会替换掉缓存中有用的数据。这些问题共同组成了数据预取中的负面效应，只有有效地控制负面效应，才能够最大限度的发挥预取器的优势。

## 2.2 数据预取负面效应控制

数据预取负面效应控制技术的相关研究，在数据预取出现的十几年内很少作为单独的研究方向。但是随着处理器应用集的更新和复杂化，数据预取难度也随之上升。通过盲目地提升预取激进度来增加预取覆盖率，不再是提高预取性能的有效方法。在有限的硬件资源内，同时提升数据预取的覆盖率和准确率难度较大。如果选择合适的预取负面效应控制方法，就可以在保证预取覆盖率提升的同时，减少预取准确率的损失。

### 2.2.1 数据预取缓冲区

为了避免预取数据对缓存造成污染，预取的数据可以被存放在高速缓存之外的缓



缓冲区中<sup>[13][34][35][36]</sup>。

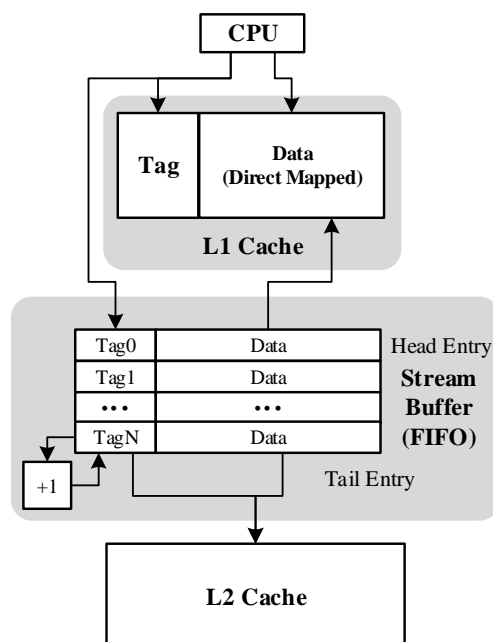


图 2.2 简单的顺序流预取器结构

经典的预取缓冲区实现思路主要来自于 Jouppi 设计的流预取器<sup>[13]</sup>。图 2.2<sup>[13]</sup>所示的这种预取器设计使用了流缓冲区来辅助预取，连续的数据会被预取到流缓冲区中直到填满为止。由于预取数据都被存放到了流缓冲区中，所以避免了预取数据对 L1DCache 的污染。流缓冲区是一个 FIFO（First In First Out）队列，当 L1DCache 发生了缺失，而流缓冲区的第一个表项被命中时，流缓冲区中相应的数据就会被存放到 L1DCache 中。在流缓冲区中的数据被使用之后，就会触发新的预取操作来重新填满流缓冲区。这样便可以保证每次需要获取新数据时，都可能在流缓冲区中找到需要的数据，从而有效地掩盖访问缺失的延迟。该设计还允许使用多个并行的流缓冲区，以便同时预取多个不同地址流，这对于同一个循环访问多个数组的情景具有很好的性能提升效果。

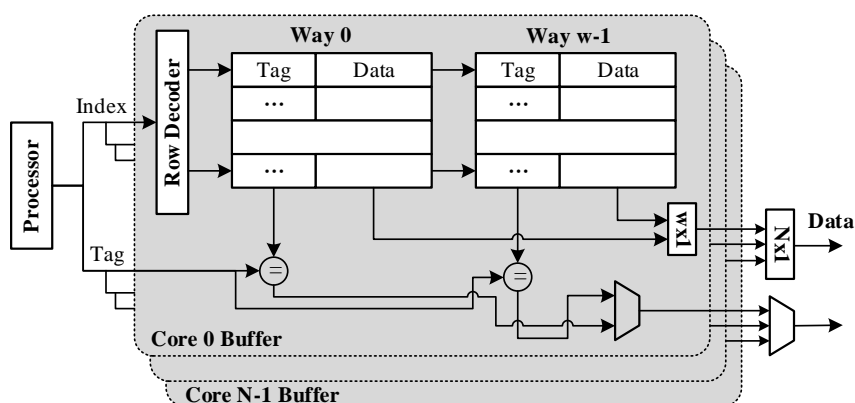


图 2.3 使用预取缓冲区的内存侧预取器

图 2.3<sup>[34]</sup>展示了 Yedlapalli 提出的一种 MSP (Memory Side Prefetcher) <sup>[34]</sup>。与流预取器不同, MSP 将缓冲区设计在 MMU (Memory Management Unit) 中。MSP 可以从内存中获取数据并将数据推送到 Cache 中, 从而避免资源争用。他们使用基于下一个缓存行的预取方案, 在行缓冲区命中时将数据存放到缓冲区中, 并立即预取下一个相邻的数据访问请求到行缓冲区。每个存储控制器单独配置的缓冲区存放预取的数据, 这样一来, 对同一行数据的后续访问就会变成 MMU 预取缓冲区命中。对预取数据的访问将不再占用内存访问带宽以及行缓冲区, 从而降低了请求的等待与响应延迟。Yedlapalli 设计的 MSP 除了减少内存访问的数量外, 还可以利用内存状态来减少行缓冲区的冲突, 从而减少未命中等待时间本身。此外, 与核心侧预取器不同, MSP 可以在内存带宽空闲时进行预取操作。通过与核心侧预取器集成可以进一步改进 MSP 的性能。其中 MSP 可以将数据提前传输到处理器中的 MMU 中。而核心侧预取器可以通过分析访问请求的规律, 预测性地访问 MSP, 并将数据传输到 Cache 中, 同时无需向处理器外的内存发送访问请求。

## 2.2.2 基于有效信息调控数据预取

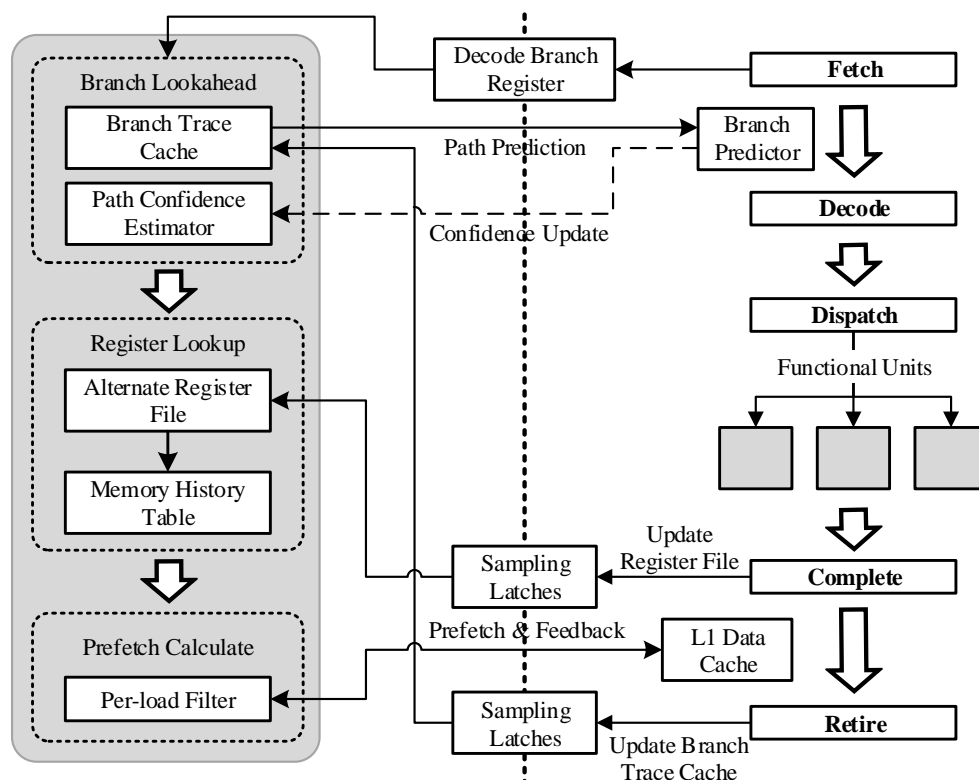


图 2.4 B-Fetch 数据预取器的结构

在发现有害预取集中于某一个预取器时，通过调整对应预取器的激进程度来避免有害预取的生成<sup>[31][37]</sup>。

Kadjo 使用分支预测器的信息，来进行高效的预取<sup>[37]</sup>，图 2.4<sup>[37]</sup>展示了这种预取器的设计结构。这种预取策略使用分支预测器获得的控制流信息，结合预测执行路径上的历史 Load 指令地址信息，生成预测执行路径上的预取指令。为了避免预取器生成的预取在 LLC 产生污染，作者还加入了对预取地址的可信度评价机制。每个 Load 指令都有一个用于记录可信度的饱和计数器，如果发现预取地址准确就会递增计数器，否则会递减计数器。当计数器的数值低于一个给定阈值时，就会关闭该 Load 指令相关的所有预取操作。

有的研究则通过直接关闭预取器，来达到避免有害预取的目的<sup>[31][37][38]</sup>。Jimenez 等人对 IBM POWER7 处理器上的预取策略进行了研究<sup>[38]</sup>。他们分别使用微测试集和标准测试集，检查了不同参数设置下，POWER7 预取器的性能和功耗。预取器的可变参数包括预取最大深度，预取步长 N，是否对固定步长大于一个缓存行的访问流进行预取，是否对 Store 指令执行预取以及是否启用预取器。由于评测程序变化较大，最佳预取器配置也会随之发生变化，因此他们提出了一种自适应预取技术，该技术可以根据测试集的特性动态配置预取器参数。该技术分两个阶段进行：在探索阶段，该技术会尝试使

用不同的参数设置预取器；在运行阶段，则会使用可以获得最大 IPC 的参数来配置预取器。由于阶段切换也可能导致 IPC 的变化，因此他们会和平均值缓冲区数据进行比较，而不是分别与各个参数配置的平均值进行比较。平均值缓冲区记录着每个参数配置下，最近一次测试阶段的 IPC。为了避免低性能参数配置对性能的影响，他们的技术会在尝试使用该配置一定次数，并且每次尝试时性能均小于给定阈值时，禁用该配置在探索阶段的测试。

Pugsley 从另一个角度出发，在只有一个预取器的准确度得到确认之后，才会对相应的预取接收并正常的处理，也得到了不错的效果<sup>[40]</sup>。

### 2.2.3 针对预取优化的缓存替换策略

从高速缓存的处理机制来说，预取之所以有害，更多情况下是因为预取的数据替换掉了即将被使用的缓存数据，或者无用的预取数据占用了额外的缓存空间。因此有一些研究针对预取适当的调整了替换策略，来避免上述情景的发生<sup>[10][41]</sup>。

Wu 发现传统的缓存替换方案对预取和非预取请求使用的是相同的处理逻辑，这使得预取往往无法提供预期的性能提升<sup>[41]</sup>。在 Wu 所设计的缓存管理方法中，可以通过调整缓存的插入顺序以及命中时在替换队列中的提升策略，来动态优化预取对缓存的干扰。他们借助预取和非预取请求的信息，来对动态引用间隔预测替换策略生成的重引用预测进行改进。他们的技术实际上是将预取与智能缓存替换方案进行了协同集成，以发挥更好的优化效果。

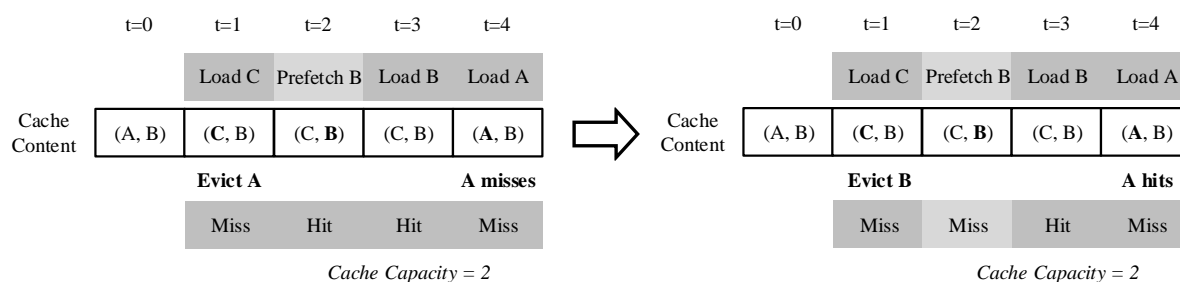


图 2.5 考虑预取情况下的最优替换

Jain 和 Lin 在最新的研究中便基于 Belady 算法设计了一种新的 Cache 替换策略<sup>[42]</sup>。即 Belady 的 MIN 算法，MIN 算法是 Belady 在 1966 年提出的最优 Cache 替换算法，即每次都会从缓存中将重用间隔最长的缓存行替换出去。但由于未来的不可预知性，该方法是不可实现的。同时由于 MIN 算法没有区分预取数据和普通数据，替换的效果并不是最优的。因为 MIN 算法并不会考虑预取到缓存中的缓存行在替换时的加权，因此在替换时可能保留即将预取的数据并替换非预取数据，而导致需求请求的访问缺失（如图 2.5<sup>[42]</sup>所示）。因此作者对 MIN 算法进行了优化，得到一种新的缓存替换算法，

该算法能够为确保预取数据在替换时拥有比非预取数据更高的优先级。在具体的实现中，该算法会优先将缓存中预取时间最晚的缓存行替换出去，为此每一个缓存行的预取时间间隔都需要被标记出来。如果找不到标记了预取间隔的缓存行用于替换，就会按照 MIN 算法选择替换的缓存行数据。

#### 2.2.4 缓解多核心的预取干扰

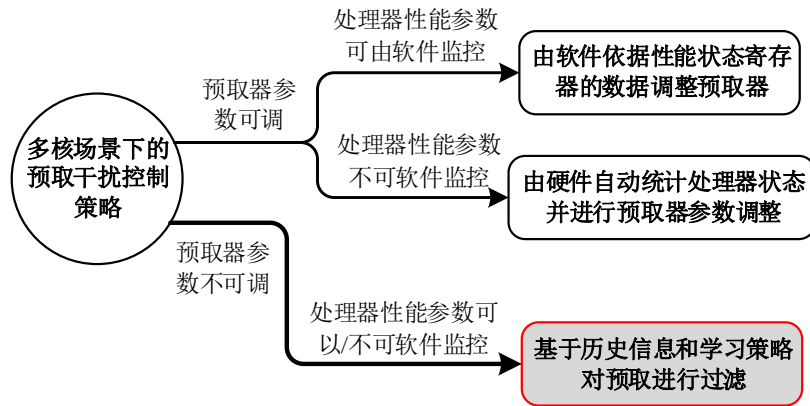


图 2.6 多核场景下的预取控制器设计分类

为了避免核间预取干扰，有些研究直接基于多核场景设计了相应的预取控制策略<sup>[17][43][44]</sup>。如图 2.6 中所示，在不同的使用场景下，预取控制策略也会有所不同。

Wu 和 Martonosi 在他们的论文中提出了一种根据预取器在 LLC 中的干扰情况，动态地打开/关闭预取器的技术<sup>[44]</sup>。他们的实验是在 Intel Core i7 处理器上展开的，其中四个不同的硬件预取器中有两个预取器，分别是 MLC（Mid Level Cache）空间预取器和 MLC 流预取器。通过软件修改状态寄存器可以对它们进行控制和调整。他们的技术利用了英特尔基于事件的精确采样功能（PEBS，Precise Event Based Sampling）收集的信息，对 LLC 未命中计数进行采样和统计。该技术与 Jimenez 等人在 IBM POWER7 中使用到的技术<sup>[22]</sup>比较相似，也分两个阶段进行。在配置阶段，会分别统计打开和关闭预取器时发生一定数量的 LLC 未命中需要的时间间隔。如果关闭预取器的情况下时间间隔较长，则表明在没有预取器的情况下，LLC 未命中发生的频率较低，因此应该在运行阶段将预取器关闭；否则预取器将会被打开。这种技术使用了较小开销，便可以有效地减轻预取对 LLC 性能地负面影响。

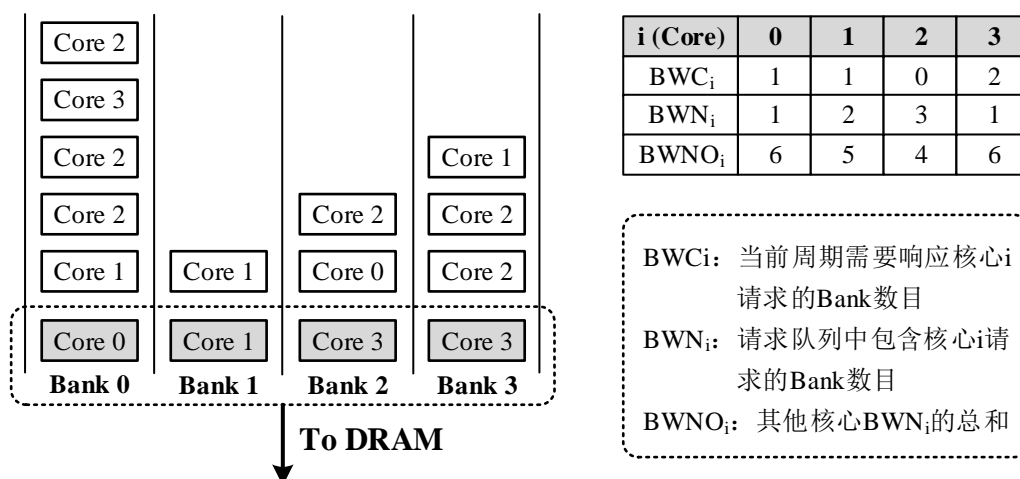


图 2.7 HPAC 统计的带宽信息

单核心场景设计的数据预取器，在多核场景下会因为带宽的浪费，导致系统性能的下降。其主要原因便是预取之间的干扰，即延迟另一个核心的预取请求响应而导致预取失效，或者导致其他核心的需求请求被延迟，甚至替换一些共享的数据从而引发需求请求的访问缺失。为了解决该问题，Ebrahimi 提出了 HPAC (Hierarchical Prefetcher Aggressiveness Control) [44]，一个多层级预取器的控制机制。图 2.7[44]展示了 HPAC 生成决策所需要的带宽信息，基于这些信息会生成优先级要高于核心内控制决策的全局控制决策。在一些情况下，全局控制决策会强制本地预取器节流来避免预取干扰；或者由本地预取器依据可信度信息主动节流，以避免生成不准确的预取。

## 2.3 数据预取负面效应控制技术对比

实际上多数预取相关的研究都会对预取器生成的预取加以限制。无论是预取器自行限流，还是对生成预取请求进行再过滤，都可以防止预取器在预取准确度下降时产生过多的不准确的预取请求，而占用带宽和污染 Cache。表 2.2 给出了目前较为常见的几种预取负面效应控制技术，它们各有优劣。有的注重于某一个负面效应的优化，有的虽然覆盖了大部分负面效应，但在单个负面效应上却不如针对该负面效应优化的技术效果好。

表 2.2 不同预取负面控制技术的效果对比

预取负面效应控制技术	提升预取 准确性	减少占 用空间	减少替 换有效 数据	减少占 用带宽	减缓多 核预取 干扰
加入预取缓冲区 <sup>[13][34][35]</sup>		√	√		
关闭预取或过滤某些 Load 指令相关的预取 <sup>[30][31][44]</sup>	√	√		√	

基于预取准确率 对预取器进行节流 <sup>[30][37][45][46]</sup>	√	√		√	
设计适应预取的 Cache 替换策略 <sup>[10][41][42]</sup>		√	√		
消除冗余的预取请求 <sup>[10][47][48]</sup>				√	
依据 Cache 访问缺失率和 IPC 过滤性 能不好时段对应的预取 <sup>[32][38][44]</sup>	√	√	√	√	
统计预取有效性，对预取进行过滤 <sup>[24]</sup>	√	√	√	√	√
基于共享带宽压力、预取状态信息对 预取进行节流与调配 <sup>[43][44]</sup>		√		√	√

每一个技术也存在各自的局限性，表 2.3 便给出了这些方法的适用 Cache 和相应的缺陷。没有哪一种方法可以全面俱到，并且没有任何硬件开销或者其他进阶负面效应。比如预取缓冲区的设计更适合于 L1Cache 这种高层级 Cache。因为 L1Cache 的容量有限，预取器生成的大量预取会严重占用 L1Cache 的空间，从而引起严重的数据污染和容量失效问题。如果将新的预取数据放到预取缓冲区中，在命中以后才转移到 L1Cache 中，就可以避免不准确预取在 L1Cache 中的危害。但是将缓冲区设计到 LLC 中并不理想，首先 LLC 的容量远大于 L1Cache，出现容量失效的比率要小得多，设计缓冲区的必要性并不大。同时如果为 LLC 设置缓冲区，缓冲区的大小一定会比 L1Cache 中设置的缓冲区大很多，这会产生较大的硬件资源浪费。相对的，对预取进行过滤往往不需要庞大的硬件资源，该技术适用于所有的 Cache 层级。

考虑到不同负面效应控制技术的局限性，近些年数据预取方面的研究常常会将多种预取负面效应控制技术进行有机的组合，以期达到更好的效果。而如何巧妙地将这些技术高效地结合在一起，在一定硬件资源限制的情况下，发挥出比单个技术更好的效果，也是相关研究所面临的主要挑战之一。

在上述预取负面效应控制技术中，借助预取有效性统计对预取进行过滤的方法，可以照顾到几乎所有的负面效应。因此本文选取了该技术作为设计与实现的基础方向，实际的设计是通过对 Bhatia E 提出的 PPF<sup>[24]</sup>，进行单核场景优化以及多核场景支持得到的，对应于图 2.6 中最下方的技术分支。相比于其他两种设计，这种设计可以在不修改 ISA（Instruction Set Architecture），不占用 CPU 的计算资源的情况下，实现细粒度的预取过滤。

表 2.3 预取负面效应控制技术的局限性

预取负面效应控制技术	适用 Cache	主要问题
------------	-------------	------

加入预取缓冲区 <sup>[13][34][35]</sup>	L1	需要额外的存储空间，并行访问功耗高
关闭预取或过滤某些 Load 指令相关的预取 <sup>[30][31][44]</sup>	L1	可能错失有效预取的机会
基于预取准确率 对预取器进行节流 <sup>[30][37][45][46]</sup>	L1-LLC	针对单核心有一定的节流效果，但是节流手段粒度太大
设计适应预取的 Cache 替换策略 <sup>[10][41][42]</sup>	L1	只能降低非有效预取的危害性，实际预取请求依旧很多，会产生带宽浪费
消除冗余的预取请求 <sup>[10][47][48]</sup>	L1	只能过滤同地址、已发出、已命中的预取请求，预取过滤很局限
依据 Cache 访问缺失率和 IPC 过滤性能不好时段对应的预取 <sup>[32][38][44]</sup>	L1-LLC	性能参数并不能完全反应出预取的影响
统计预取有效性，对预取进行过滤 <sup>[24]</sup>	L1-LLC	统计有效性需要一定的存储空间记录替换信息，过滤操作会产生额外的延迟
基于共享带宽压力、预取状态信息对预取进行节流与调配 <sup>[43][44]</sup>	L1-LLC	可以平衡带宽压力，但可能会屏蔽掉准确的预取请求

## 2.4 本章小结

本章对现有的数据预取负面效应控制技术进行了分析。本章节首先对当前主流数据预取器进行了汇总和分类。然后以数据预取缓冲区、基于有效信息调控数据预取、针对预取优化缓存、避免不同核心之间的干扰的替换策略为例，对国内外学者在数据预取研究中提出的预取负面效应控制技术的进行了介绍。最后对不同的数据预取负面效应控制技术的优化效果和适用范围进行了归纳和分析。并选择了通过统计预取有效性，对预取进行细粒度过滤的方法作为本次研究的基本方向。



## 第三章 数据预取控制器的设计与实现

本章首先介绍了 Gem5 模拟器中的 Cache 结构设计，以及对预取的处理方式。然后从预取分类统计模块、预取有害性统计模块、预取过滤器模块三个方面，介绍了面向多核的数据预取控制器的设计与实现。除了介绍基本的结构设计外，本章节还对预取层级变更有效实现、扩展的训练策略，时钟周期准确的预取无效化传递等设计细节进行了详细的介绍。最后对数据预取控制器的硬件开销进行了分析和评估。

### 3.1 Gem5 的 Cache 设计分析

Gem5 是一个时钟周期准确的体系结构模拟器，它拥有可配置的乱序执行 CPU 模型及完整的存储层次结构模拟系统，支持多种 ISA。相比于近期论文<sup>[19][20][22]</sup>中经常使用的，DPC-2<sup>[49]</sup>/DPC-3<sup>[50]</sup>官方指定的 ChampionSim<sup>[51]</sup>模拟器，Gem5 具有更强大的可配置性以及更精确的模拟结果。因此本文选择 Gem5 作为数据预取控制器的实现平台，以获得更加准确的性能评测与分析结果。为了能够在 Gem5 中设计数据预取控制器，首先需要理解 Gem5 模拟的硬件设计对数据预取的处理过程。本节主要对 Gem5 中的 Cache 结构和数据预取处理方式进行了简要的分析。

#### 3.1.1 Gem5 的 Cache 结构设计

Gem5 的存储器系统支持自定义配置，不同层次的存储结构借助 XBar 连接起来，而 XBar 和 Cache 之间借助 Port 进行数据请求通信。图 3.1 以使用 3 级 Cache 的双核处理器为例展示了可配置 Cache 在 Gem5 中的连接方式。

其中 Port 分为两类，分别是 MasterPort 和 SlavePort，MasterPort 连接到低层级的存储结构，SlavePort 连接到高层级的存储层次或处理器核心。MasterPort 可以发送请求数据或者接收请求的反馈数据，而 SlavePort 可以接收请求数据或者发送请求的反馈数据。所有的请求和相关数据都是通过 Gem5 封装的 Packet 类型实现的。

当 Cache 从 SlavePort 接收到来自高层级 Cache 或者处理器核心的请求时，会首先计算对应的 Packet 到达 Cache 需要的时间保证时序模拟的准确性，并依据请求的类型查找 Cache 中匹配的缓存行数据。如果请求是一个写回操作，则会将需要写回的数据写入到当前 Cache 中，并依据替换数据的状态位，生成传递给下一级 Cache 的写回操作。如果请求是一个读操作，并命中了 Cache 中的缓存行，则会生成读请求反馈，由本层级 Cache 的 SlavePort 传回上级 Cache 或处理器核心，同时将对应的缓存行的预取属性位重置。没有命中 Cache 中数据，但是命中了 MSHR (Miss-status Handling Registers)，

读请求会合并到现有的 MSHR 表项中, 否则会分配新的 MSHR 表项。如果分配新 MSHR 表项导致 MSHR 表项用尽, 当前 Cache 就会进入阻塞状态, 在新的 MSHR 表项被释放前将无法响应任何上级 Cache 或者处理器核心传递下来的请求。如果处理周期结束时, 存在可用的 MSHR 表项并且开启了预取器, 则会从预取器结构中获取一个新的预取请求进行处理。

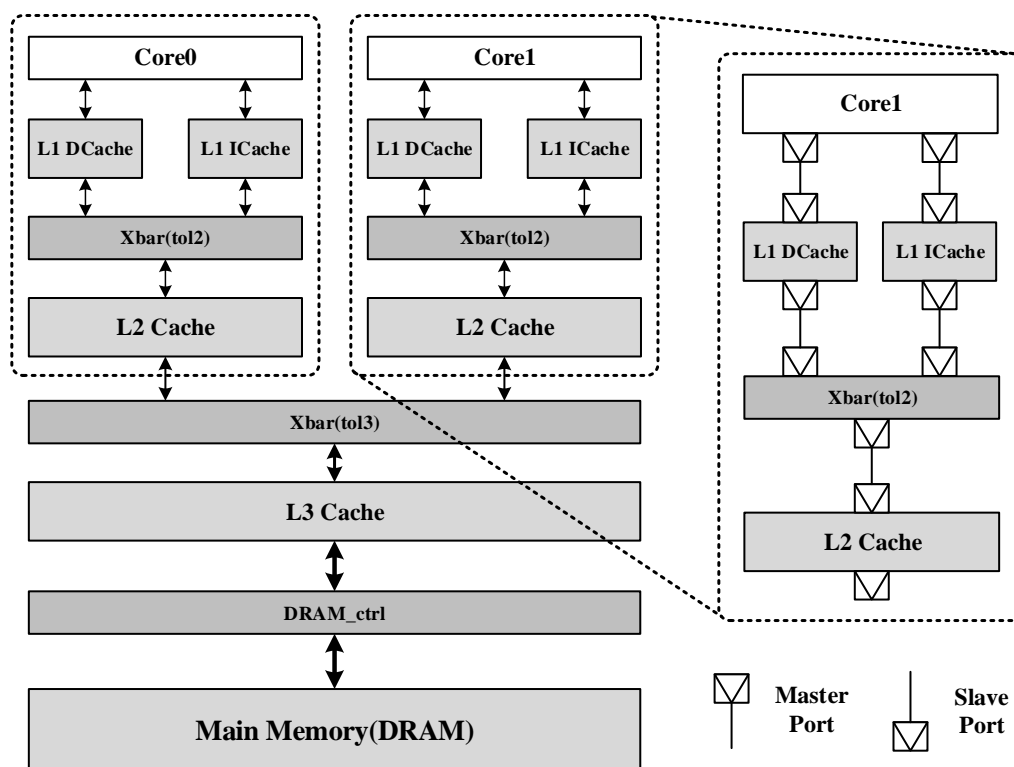


图 3.1 Gem5 中双核处理器的存储层次结构

当 Cache 从 MasterPort 接收到低层级 Cache 或主存的请求反馈时, 会忽略发生错误的请求反馈。Cache 收到的请求反馈大多是读取请求的反馈, 处理读取请求的反馈时会寻找替换的缓存行, 并执行整行数据的写入, 更新对应缓存行的状态。为了保证时序的准确性, 在进行缓存行替换的时候, 填充的缓存行数据在特定的写入延迟后才会被标记为准备就绪的状态。在读取数据填充处理完毕后, 就会按照合并的顺序处理 MSHR 中记录的每一个上级 Cache 或处理器核心的请求, 生成请求反馈并将反馈请求由 SlavePort 发送给上级的 Cache 或者处理器核心。MSHR 表项在处理结束后就会被释放, 如果当前 Cache 因为 MSHR 表项不足而处于阻塞状态, 就会清除阻塞状态。如果进行替换操作时没有找到用于替换的缓存行, 则会将请求数据缓存起来, 在处理完 MSHR 表项相关的请求反馈之后, 立即为相关数据执行写回操作以保证数据的正确性。

### 3.1.2 Gem5 的请求传输设计

Gem5 借助 Port 实现了不同层级 Cache 之间的互联, 而 Port 如何对请求或者请求

反馈执行发送和接收操作，决定了 Gem5 如何在存储层次结构中处理 Cache 内互连通信。图 3.2 详细地展示了 Cache 和 Port 中的数据传递情况。由于一个周期内可能生成多个请求反馈，而一组连接在一起的 MasterPort 和 SlavePort 每个周期最多只能传递一个请求或者请求反馈。因此每一个 Cache 都设置了一个基于 FIFO 队列设计的需求反馈队列（RespQueue）来缓存反馈信息。

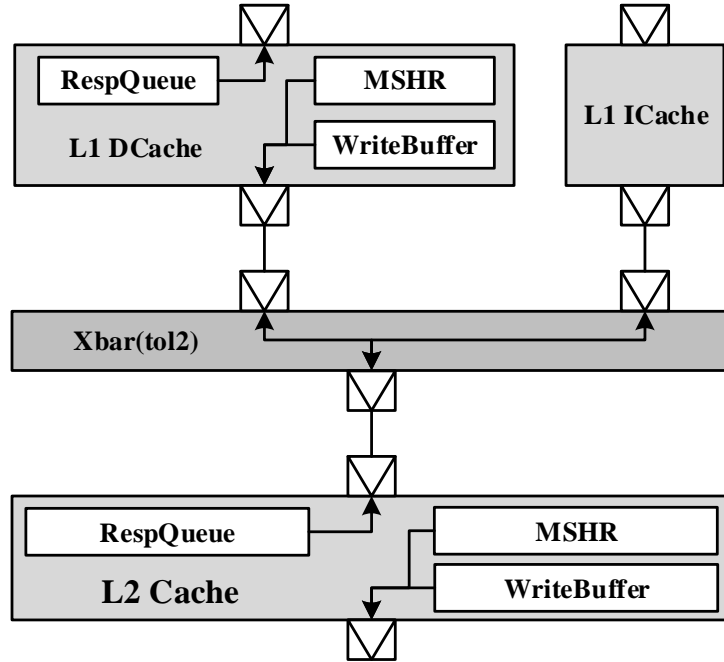


图 3.2 Gem5 中基于 Port 的连接结构

当 Cache 发现有新的请求反馈可以发送，确定其实际完成时间后，就会向 SlavePort 发送请求反馈。请求反馈会按照完成时间顺序被插入到请求反馈队列中，或者优先按照收到请求反馈的顺序插入。每一个周期 SlavePort 都会尝试调度并发送位于请求反馈队列顶端的需求反馈，到上级 Cache 或者处理器核心的 MasterPort 中。如果当前这一组 Port 连接没有被占用，就能够成功的将请求反馈通过 XBar 发送给上层的 Cache 或者处理器核心。

同时，Cache 在每一个周期都会尝试从写缓冲区（WriteBuffer）或者 MSHR 中找出一个准备就绪的请求，并发送到低层级的 Cache 或主存中。在写缓冲区填满的时候会优先处理写缓冲区存放的请求，否则会优先处理 MSHR 中准备就绪的请求。或者说 Cache 会优先发送可以让 Cache 阻塞状态解除的请求。如果发送请求所使用的 MasterPort 当前没有被占用，则请求可以成功发送。请求地成功发送可以直接释放写缓冲区表项，但不能释放 MSHR 表项，这也是在写缓冲区没有空余表项时会得到优先处理主要原因。请求发送失败除了因为发送请求的 MasterPort 连接正在被占用以外，更多情况是因为低层级的 Cache 或者主存处于阻塞状态。

Cache 在通过 Port 处理请求和请求反馈时有着比较大的区别。虽然处理请求反馈时也存在阻塞的情况，但是和处理请求的阻塞并不一样。待发送请求的信息对处理器核心来说存放在 LSQ (Load Store Queue) 中，对 Cache 来说存放在 MSHR 和写缓冲区中。只要请求没有发出去，对应请求的信息就会一直存在于这些结构里面。这些结构起到了缓存请求信息队列相同的效果，但是他们支持着更多地，诸如读数据旁路传递、写缓冲、读失效合并等可以有效降低访存开销的关键技术。

相对的，处理请求反馈时使用了请求反馈队列缓存请求反馈信息。缓存的请求反馈会对缓存的反馈信息在进入队列时会进行排列，一般会按照完成时间的先后顺序进行排列，有时候也会要求按照收到请求的顺序排列。虽然在 Cache 看来，请求反馈是立即完成的，但实际上请求反馈只是暂时缓存到了请求反馈队列中，在合适的时机才会被传递到上层的 Cache 或者处理器核心中得到处理。由于请求反馈的数量一定不会超过请求的数量，所以 Gem5 并没有将反馈队列大小作为一个可配置的硬件结构参数。

### 3.1.3 Gem5 的数据预取设计

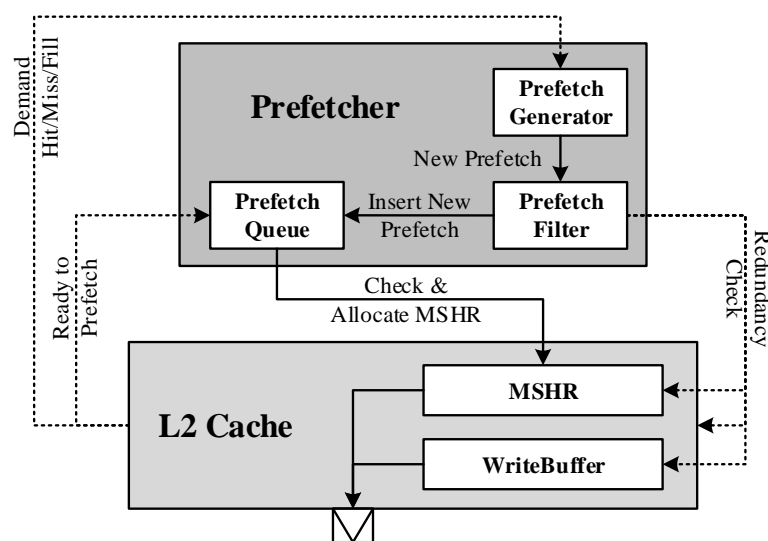


图 3.3 Gem5 中 L2Cache 的预取器结构

Gem5 中每一个 Cache 都可以配置一个私有预取器，图 3.3 以 L2Cache 中的数据预取器为例展示了 Gem5 中预取器的设计结构。Gem5 并没有对指令预取器和数据预取器进行严格的区分。在 L1Cache 中严格区分指令预取和数据预取是有意义的，但是更低层级的 Cache 中同时存放着指令数据和非指令数据，区分指令数据和非指令数据的意义就变得很小了。因此 L2Cache 的数据预取器除了预取非指令数据外，还被允许预取指令数据。

Gem5 的预取器主要由预取请求队列 (Prefetch Queue)、预取过滤器 (Prefetch Filter) 和预取生成器 (Prefetch Generator) 三个主要结构组成。它们分别负责预取请求的缓存、

冗余过滤和生成。

不同数据预取器的主要区别在预取生成器的结构设计上。每当预取器所属的 Cache 出现访问命中、访问缺失或数据填充时，相关信息都会发送到预取生成器中用于生成新的预取。如果新的事件触发了预取训练，并生成了新的预取，那么预取就会被传送给预取过滤器进行初步过滤。初步过滤会同步检查预取请求队列、Cache、MSHR 和写缓冲区，如果预取请求在这四个结构中任何一个中已经存在就会被忽略。只有没有被初步过滤的预取请求才可以被插入到预取请求队列中。预取请求队列是一个典型的 FIFO 队列，队列中最早插入的预取请求会优先发送给 Cache 处理。如果在向预取队列中插入新的预取请求时，预取请求队列已经没有空位，那么新的请求就会将队首的，即最陈旧的预取请求挤掉。因此预取请求队列存放的一定是最近生成的预取请求。

在介绍 Cache 设计时，曾经提到 Cache 只有在 MSHR 和写缓冲区都没有准备就绪的可以发送的请求时才会从预取器获取预取请求。Cache 在获得预取请求之后并不会立即为它分配 MSHR。由于预取请求从生成到传递给 Cache 之间存在一段时间隔，期间预取相关的数据或请求，可能已经存在于 Cache、MSHR 或者写缓冲区中。因此 Cache 会再次进行冗余性检查，之后才会为没有发生冲突的预取请求分配 MSHR。

为了区分预取数据和由需求请求获取的数据，每一个缓存行都会设置单独属性位标记数据是否是一个预取数据。这个属性在预取的数据达到并进行填充时设置，并在预取数据被命中后清除。然而实际上，当预取的数据被预取请求命中后，其实际属性依旧是预取。因此经过本文的修改，Gem5 在发生预取数据命中时，只有发生命中的请求并非预取请求，才会清除缓存行的预取属性。

#### 3.1.4 Gem5 中基于步长的预取器

表 3.1 基于固定步长的数据预取器默认配置

基于固定步长预取器的参数	参数数值
预取请求队列长度	32
可信度最大值	7
可信度最小值	0
生成预取的可信度阈值	4
可信度初始化数值	3
PCTable 的 Set 数目	16
PCTable 组相联度	4
预取深度	16

本文研究选择的基准数据预取器是基于固定步长的预取器。正如表 1.2 展示的那

样，当前大部分主流的商用处理器使用的依旧是那些传统的数据预取器，而基于固定步长的数据预取器，正是其中最常用的数据预取器之一。

在 Gem5 中，基于固定步长的数据预取器包含了一个名为 PCTable 的主体结构，该结构记录了最近访问命中或访问缺失的相关信息。PCTable 基于组相联结构设计，使用访问 PC 的哈希值索引，使用完整的 PC 作为 Tag，在替换算法上使用的是实现较为简单的随机替换。PCTable 中的表项记录了最近访问对应的指令 PC、最近一次的历史访问地址、步长、可信度四个数据。

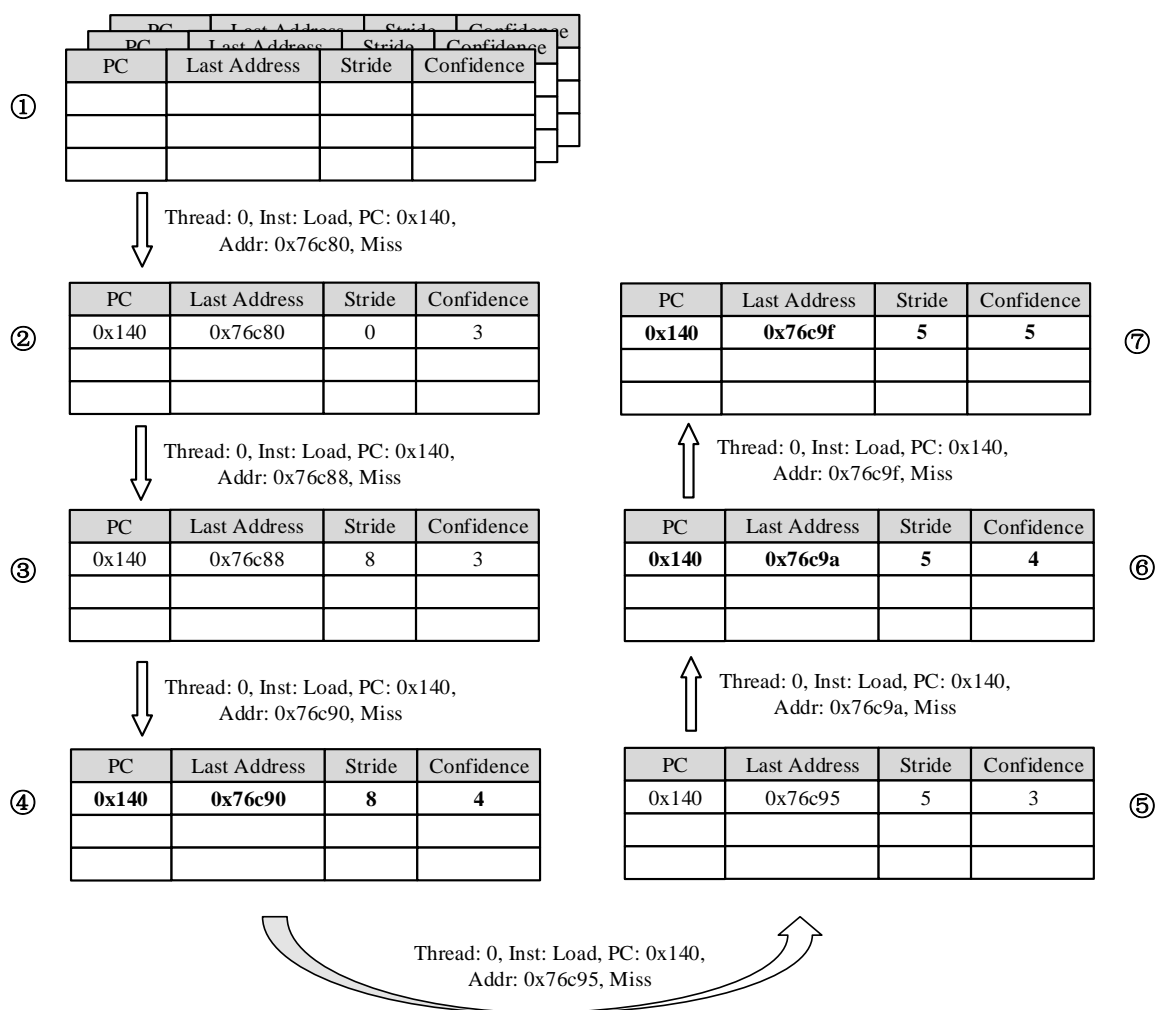


图 3.4 基于步长预取器的训练与预取过程

图 3.4 简要展示了 Gem5 中基于固定步长的数据预取器的训练和预取过程，表 3.1 给出了预取器的默认配置参数。如果一个新的访问没有对应的指令 PC<sup>①</sup>，那么它会被忽略。如图 3.4 中①所示，数据预取器会给每一个线程上下文建立独立的 PCTable 以避免不同线程之间的预取训练干扰。当一个新的访问命中或者访问缺失出现时，预取器

<sup>①</sup> 这类指令一般是来自于 Cache 内部生成的访问请求，比如脏数据的写回请求

会查找 PCTable 中是否有对应的表项, 如果没有则会按照初始化标准, 使用空表项或者替换一个 PCTable 表项来存放新的信息。如果命中了 PCTable 的有效表项, 则会依据新的访问地址和表项中记录的历史访问地址确定一个新的步长。当新的步长和原来表项的记录步长一致时, 对应表项的步长可信度就会递增; 如果步长发生了变化, 那么该表项的可信度数值就会递减。当这个可信度数值降低到可信阈值以下时, 旧步长就会被新步长所替换。

如果命中的 PCTable 表项中可信度达到了可信阈值, 比如图 3.4 中的④、⑥和⑦, 就会依据设置的预取深度和触发训练的访问地址生成一组预取请求。对 L1DCache 来说, 访问地址往往不是缓存行对齐的。但如果预取的步长小于缓存行的大小, 预取器就会以缓存行的大小作为实际的步长进行预取。强制将预取步长对齐到缓存行大小, 也会导致实际步长小于缓存行大小时, 产生过多的预取请求。

## 3.2 数据预取控制器设计概述

本小节会首先对数据预取控制器中的基本概念进行介绍, 然后对数据预取控制器的整体结构进行简单的说明。

### 3.2.1 基本概念

本文对 Gem5 模拟器中的一些基本设置进行了修改, 因此在论文中会出现一些新的概念。为了保证后续论文描述的准确性, 接下来会对其中一些关键的概念进行解释和说明。而相应的实现策略和处理流程, 会在后面的章节中进行更为详细的描述。

为了区分存储系统中不同事件对应的数据类型, 本文对数据类型进行了划分。每一个请求或者 MSHR 的属性都会和相应的数据类型挂钩:

- 1) **Prefetch**: 预取类型, 表明该数据是由一个硬件预取器生成的预取请求处理的; 处理器核心执行并发出的软件预取请求不属于该类型
- 2) **Demand**: 来自处理器核心的访问请求的数据类型均为 Demand; 来自 Cache 的部分请求类型为 Demand, 这些请求负责数据写回和 Cache 数据一致性的维护。这一类请求必须得到响应, 否则会导致程序的执行错误
- 3) **NullType**: 空类型, 用于特殊情况的处理, 比如缺失时目标类型被标记为 NullType, 表明访问请求申请了一个新的 MSHR; 数据填充时目标类型被标记为 NullType, 则表示填充数据时使用的是空缓存行。
- 4) **Pending Prefetch**: 处理中的预取类型, 当一个存放预取请求的 MSHR 访问请求已经发送出去并正在被处理时, 数据类型就会由 Prefetch 变为 Pending Prefetch

本文在 Gem5 中添加了对数据预取目标层级变更的支持，比如 L2Cache 的预取请求最后可以将数据放到 L3Cache 中，也可以放到 L1DCache 或 L1ICache 中。针对不同预取层级变更情况，给出下面三个概念：

- 1) **提级预取**：目标 Cache 层级高于发出预取请求 Cache 层级的预取
- 2) **平级预取**：目标 Cache 层级等于发出预取请求 Cache 层级的预取
- 3) **降级预取**：目标 Cache 层级低于发出预取请求 Cache 层级的预取

### 3.2.2 整体结构设计

为了进一步优化数据预取器在单核场景及多核场景的性能，本文基于 Eshan Bhatia 设计的 PPF 设计了面向多核的数据预取器控制器，来进一步优化数据预取器的性能。其中图 3.5 展示了本文设计的数据预取器控制器的整体结构，控制器主要由数据预取统计分析模块、数据预取有害性统计模块和数据预取过滤器模块三个部分组成。

其中数据预取统计分析模块用于对预取请求的行为进行详细分析和统计，该结构支持多达 17 中预取特性的统计。由于仅仅用于实验分析，该结构无需硬件实现。数据预取有害性统计模块会对每一个预取的有害性进行统计，并对有害预取执行相应的惩罚训练。数据预取过滤器模块是数据预取器控制器的核心结构，该结构在优化后的 PPF (PPF+) 的基础上扩展得到，结合了感知器学习原理和预取全局信息对预取进行过滤。该结构会提取每一个预取不同的特征信息，比如预取地址、预取深度等等，并为每一个特征设置一个权重。通过对不同预取事件训练来，动态的改变权重大小，并基于权重对预取进行过滤。

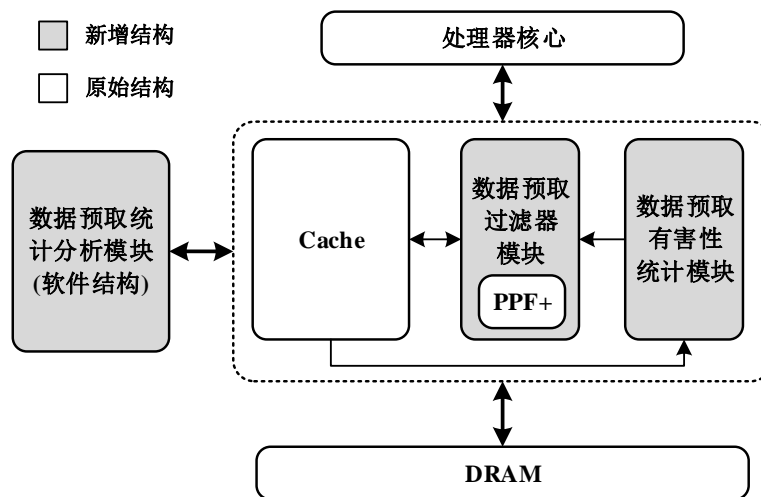


图 3.5 数据预取控制器的整体结构

借助这三个模块，本文不仅可以对预取干扰的原因进行更深层次的分析，还可以在单核或多核场景下进一步提升数据预取器的性能。下面的三个章节会对三个主要模块的设计方法进行更为详细的说明。



### 3.3 数据预取统计分析模块

为了对预取行为进行更为有效且精确的分析，本文设计并实现了数据预取统计分析模块对预取的详细信息进行统计和分析。该模块仅用于生成分析用的实验数据，因此无需硬件实现。

#### 3.3.1 数据预取的分类

主流的数据预取研究依旧将预取准确性、时效性、覆盖率等作为评判预取的标准，但是很少有研究对预取的实际行为进行分类和分析。本文结合了其他数据预取的相关研究，对数据预取的行为进行了详细的划分，并基于行为划分对数据预取进行了详尽的分类。通过对不同分类的数据预取数量进行比较，可以更加有效的分析出预取器设计的优势或缺陷。

在单核场景下，数据预取的分类比较单一，大体上可以将单核场景下的预取分为四类：

- 1) **有效预取**：预取数据在被替换之前被访问过，而预取替换的数据在预取数据被替换之前没有被访问过。
- 2) **无效预取**：预取数据在被替换之前未被访问到，同时预取替换的数据在预取数据被替换之前也没有被访问到。
- 3) **不定性预取**：预取数据在被替换之前被访问过，同时预取替换的数据在预取数据被替换之前也被访问过且没有命中。
- 4) **有害预取**：预取数据在被替换之前未被访问过，而预取替换的数据在预取数据被替换之前被访问过且没有命中。

单核场景下的预取分类较为简单。在多核场景下，由于预取的影响对象都有所变化，因此相应的分类会更加详细。为了获取预取的分类，需要使用量化的数据进行比对。因此每一个预取都会拥有 4 个量化数据指标（核心 A 指代发出预取请求的核心）：

- 1) **核内有效性 ( $\beta_{self-use}$ )**：核心预取数据对自身的有效性，当核心 A 预取数据被核心 A 命中时，核内有效性数值就会递增。
- 2) **核内有害性 ( $\beta_{self-harm}$ )**：核心预取数据对自身的有害性，当核心 A 预取数据填充到 Cache 时替换的数据，在预取数据被替换前被核心 A 访问且发生了访问缺失，核内有害性数值就会递增。
- 3) **核间有效性 ( $\beta_{other-use}$ )**：核心预取数据对其他核心的有效性，当核心 A 预取数据被核心 A 以外的其他核心命中，核间有效性数值就会递增。
- 4) **核间有害性 ( $\beta_{other-harm}$ )**：核心预取数据对其他核心的有害性，核心 A 预取数据填充到 Cache 时替换的数据，在预取数据被替换前被核心 A 以外的其他核心访问，并且发生了访问缺失，核间有害性数值就会递增。

$$\delta_{self-abs} = \beta_{self-use} - \beta_{self-harm} \quad (3-1)$$

$$\delta_{other-abs} = \beta_{other-use} - \beta_{other-harm} \quad (3-2)$$

表 3.2 多核场景下的预取分类条件

多核场景下的预取分类	分类条件
多核有效预取	$\delta_{self-abs} \geq 0; \delta_{other-abs} > 0;$ $\delta_{self-abs} \leq \delta_{other-abs}$
单核有效预取	$\delta_{self-abs} > 0; \delta_{other-abs} \geq 0;$ $\delta_{self-abs} > \delta_{other-abs}$
无效预取	$\delta_{self-abs} = 0; \delta_{other-abs} = 0;$
自私的预取	$\delta_{self-abs} > 0; \delta_{other-abs} < 0;$
无私的预取	$\delta_{self-abs} < 0; \delta_{other-abs} > 0;$
多核有害预取	$\delta_{self-abs} \leq 0; \delta_{other-abs} < 0;$ $\delta_{self-abs} \geq \delta_{other-abs}$
单核有害预取	$\delta_{self-abs} < 0; \delta_{other-abs} \leq 0;$ $\delta_{self-abs} < \delta_{other-abs}$

通过计算核内有益性数值和核内有害性数值的差值，可以得到**核内绝对有效性** ( $\delta_{self-abs}$ )。绝对有效性可以是正值也可以是负值，为负值时表示有害性高于有益性。同理也可以借助核间有效性数值和核间有害性数值，计算得到**核间绝对有效性** ( $\delta_{other-abs}$ ) 的大小。依据这些数据和表 3.2 中展示的条件，便可以将多核场景下的预取请求进行更详尽的分类。表 3.2 中展示的 7 种分类囊括了所有可能的预取情况，忽略了单项指标的大小，更侧重于不同量化指标之间的关系。本文设计与实现的数据预取统计分析模块便支持了该表中所展示的预取分类。

### 3.3.2 预取统计分析模块的设计与实现

为了实现预取的有效性统计，本文在 Gem5 中添加了新的预取统计分析模块 (Prefetch Monitor)。图 3.6 展示了预取统计分析模块的基本结构设计。预取统计分析模块只提供统计分析相关的功能。由于该模块仅用于生成分析所需要的实验数据，并不需要支持硬件设计与实现。因此预取统计分析模块包含的相关结构不需要考虑存储开销的问题，可以按照最理想的方式实现。为了区分不同的预取请求，并对它们逐个进行统计，本文为 Gem5 的每一个预取器添加了唯一全局 ID，来区分预取请求的来源。预取统计分析模块则会使用预取的地址、发出预取的预取器 ID 和预取生成的时间哈希得到的数值，作为预取请求的全局唯一 ID。

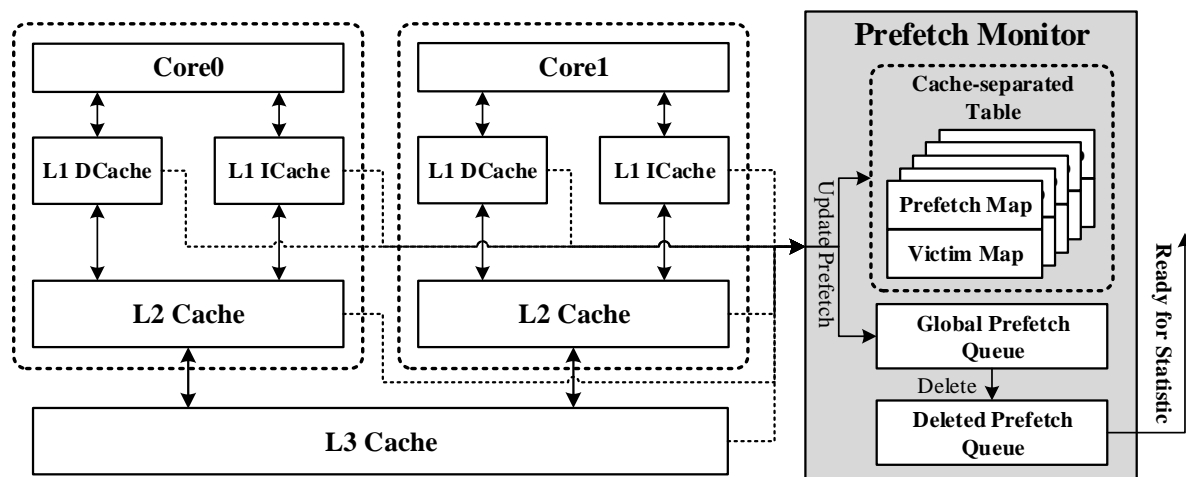


图 3.6 数据预取分类统计模块结构

预取统计分析模块中的本地预取信息记录表（Cache-separated Table）会为每一个可能存放预取数据的 Cache 设置一组映射表格，用于记录预取信息。其中预取信息映射表（Prefetch Map）记录了与当前 Cache 中存放的预取数据相关的所有预取请求 ID，而预取替换信息映射表（Victim Map）记录了与当前 Cache 替换数据相关的所有预取请求 ID。两者均通过地址映射进行查找，每一个地址可以同时映射关联到多个预取信息。全局预取信息队列（Global Prefetch Queue）存放着所有有效的预取请求信息实体。一个预取请求只有在数据至少存在于一个 Cache 中，并且对应数据拥有预取属性时，才可以出现在全局预取信息队列中。最近从全局预取信息队列中删除的预取信息会被缓存到统计就绪预取信息队列（Deleted Prefetch Queue）中，之后被用来更新统计数据。

在本文的实现中，预取统计分析模块只有一个全局唯一实体，它和所有当前处理器中的 Cache 相连。每一个 Cache 中发生的访问命中、访问缺失等关键事件都会通知预取统计分析模块进行相关的统计数据更新。表 3.3 展示了预取统计分析模块针对不同事件的处理流程，下面对其中的特殊情景进行说明：

- a) 当需求请求命中预取数据时，对应预取的有效性数值就会递增。由于一个需求请求的源头可以来自多个不同的核心，因此对应的有效性数值可能会递增多次。而本文为了确定一个需求请求的所有源头，还在 Gem5 模拟器中添加了请求源相关信息及相应的合并操作。
- a) 当需求请求命中了预取数据，那么无论是在 Cache 中还是预取统计分析模块的相关记录中，其预取属性都会被清除。因此会删除该预取在当前 Cache 对应的本地预取信息记录表中，预取信息映射表和预取替换信息映射表中的记录。此外为了统计到更加真实的信息，需求请求命中导致的当前级别 Cache 预取数据的预取属性删除，会触发预取无效化操作的向下传递。预取无效化传递的相关内容会在之后的章节中进行说明。

表 3.3 预取分析统计更新流程

事件	请求类型	目标类型 <sup>①</sup>	处理
访问命中	Demand	Prefetch	更新预取命中计数 <sup>[a]</sup> ; 删除预取 <sup>[b]</sup>
	Prefetch	Prefetch	更新预取命中计数; 预取信息合并 <sup>[c]</sup>
	Prefetch	Demand	按情况删除预取 <sup>[d]</sup>
访问缺失	Demand	Prefetch	尝试更新有害性计数 <sup>[e]</sup> ; 删除预取
	Demand	Demand/NullType	尝试更新有害计数 <sup>[e]</sup>
	Prefetch	Prefetch	按情况删除预取 <sup>[f]</sup>
	Prefetch	Pending Prefetch	按情况删除预取 <sup>[g]</sup>
	Prefetch	NullType	按情况添加预取 <sup>[h]</sup>
数据填充	Demand	Prefetch	删除预取
	Prefetch	Demand	添加预取替换信息 <sup>[i]</sup>
	Prefetch	Prefetch	按情况记录预取替换信息 <sup>[j]</sup> ; 删除预取
缓存行无 效化	Prefetch	—	删除预取

- b) 预取请求命中了预取数据，并不会更新有效性数值。但是预取数据命中预取，就意味着当前 Cache 中的预取数据之后所产生的任何事件，应该同时和这几个预取关联起来，进行同步的更新。在预取信息合并到当前 Cache 级别的预取信息映射表和预取替换信息映射表中以后，相关的合并操作也会向下传递，其传递方法和无效化传递的方式相同，均基于时钟周期准确的通信操作实现。
- c) 如果预取请求命中 Demand，且该预取请求的目标 Cache 层级恰好为当前的层级，那么这个预取不会有任何用处，相关的信息会被立即删除。
- d) 如果一个需求请求发生了访问缺失，同时在预取替换信息映射表中查找到了对应的记录，就会更新相关预取的有害性数值。由于一个需求请求的源头可以来自多个不同的核心，因此对应的有害性数值可能会递增多次。

<sup>①</sup> 对于访问命中，目标类型指的是命中缓存行的属性；对于访问缺失，目标类型指的是 MSHR 中表项的属性；对于数据填充，目标类型指的是被替换缓存行的属性

- e) 预取请求发生了访问缺失，但是命中了 MSHR 并合并到了其中预取的表项中。这时候会依据不同层级预取在 MSHR 的合并规则，判断哪一个预取被无效化，并删除无效化预取的相关信息。预取在 MSHR 中的合并规则会在后续预取层级变更的相关章节中进行说明。
- f) 预取请求发生了缺失，但是命中了 MSHR 正在处理中的预取表项。如果新的预取请求是一个相对当前 Cache 的降级/平级预取，则该预取会被无效化，并删除相关的预取信息。
- g) 预取请求发生了缺失，分配了一个新的 MSHR 表项缓存请求，同时检测到该预取正是当前 Cache 预取器发出的预取请求，则会向全局预取信息队列中添加一个新的预取信息跟踪该请求。
- h) 当预取请求完成，开始进行数据填充并替换了 Demand 类型数据，则会添加替换信息。除了向当前 Cache 对应的预取信息映射表中添加信息外，还会在对应的预取替换信息映射表中添加替换数据的信息。
- i) 如果预取请求完成并进行数据填充时，替换的是预取数据，除了删除被替换的预取相关信息外，还会更新预取请求的替换信息。为了确保统计的真实性，如果预取数据替换预取数据，预取替换信息映射表中记录的并不是被预取替换的数据地址，而是继承之前预取替换的 Demand 数据地址。

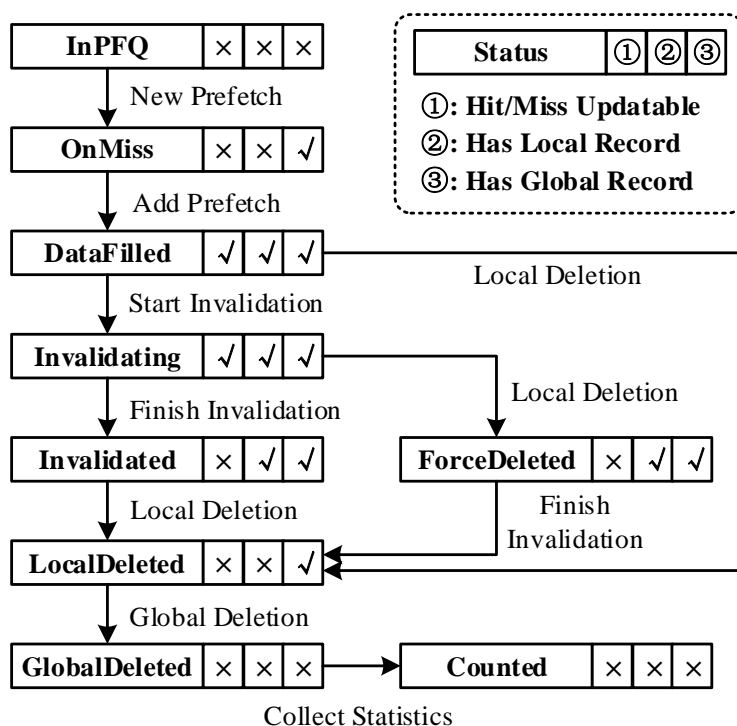


图 3.7 预取在预取统计分析模块中记录状态的变化

为了支持时间维度的信息统计，预取统计分析模块设置了一个统计数据更新周期。

在每一个更新周期结束的时候，都会收集上一个更新周期内信息进行输出时间维度下的统计结果，同时统计就绪预取信息队列中缓存的预取信息也会被统计然后彻底的删除。

图 3.7 展示了一个预取请求相关信息在预取统计分析模块中整个生命周期的变化过程。其中每一个状态都有三个子状态：第一个子状态表示当前的预取是否可以进行有效性或者有害性的相关更新；第二个子状态表示预取信息在本地预取信息记录表中是否存在记录；第三个子状态表示预取信息是否在全局预取信息队列中存在记录。当预取尚未分配 MSHR 表项的时候，处于 In 预取请求队列状态，没有任何相关的信息记录。在分配了 MSHR 表项之后，预取统计分析模块就会为预取添加信息记录，并存放到全局预取信息队列中，进入 OnMiss 状态。在预取请求正确地完成，并进行了数据替换与填充后，进入 DataFilled 状态，预取信息才会被记录到本地预取信息记录表中。如果之后直接对该预取进行删除，则本地预取信息记录表中的记录就会被删除。

Gem5 只有在当前预取数据被命中，或者预取数据被无效化后才会清除预取属性，同时属性清除只会在当前 Cache 中发生。但是对于因为命中导致的预取属性清除情况，即便没有预取，需求请求的访问缺失依旧会导致低层级 Cache 填充对应的数据，因此本文设计了预取无效化向下传递的操作以获得更真实的统计结果。如果遇到预取无效化向下传递操作，比如 L2Cache 中预取数据 A 因为被需求请求命中失去了预取属性，那么预取统计分析模块在经过 L3Cache 的通信延迟对应的时间后，也会将 L3Cache 中的数据 A 预取无效化。在无效化通信计时开始后，预取会进入 Invalidating，预取无效化计时状态，期间保留本地记录，依旧可以进行有效性或者有害性的更新。如果在无效化计时结束前，本地 Cache 便对数据 A 的预取属性进行了清除或数据 A 被无效化，则会进入 ForceDeleted 状态，并在无效化计时结束之后完成操作进入 LocalDeleted 状态。否则预取无效化计时结束之后，就会进入 Invalidated 预取无效状态，这时候本地依旧会保留简要的记录，以便保证无效化的正确完成，但不在支持预取有效性和有害性的更新。只有在当前 Cache 真正触发预取属性清除操作或预取数据被无效化后，才会进入 LocalDeleted 状态。

进入 LocalDeleted 状态的预取，在当前 Cache 对应的本地预取信息记录表中是没有任何记录的。但由于预取数据可能同时存在与其他层级或者其他核心的 Cache 中，该信息并不会立即从全局预取信息队列中删除。当且仅当该预取在所有 Cache 中的数据都被无效化，或预取属性都被清除时，才会触发 Global Deletion，将预取信息从全局预取信息队列中删除。删除的预取信息会缓存在统计就绪预取信息队列中，在每一个统计数据更新周期结尾进行统计更新后，这些预取信息才会被完全删除。

## 3.3.3 预取统计分析模块的统计变量及更新策略

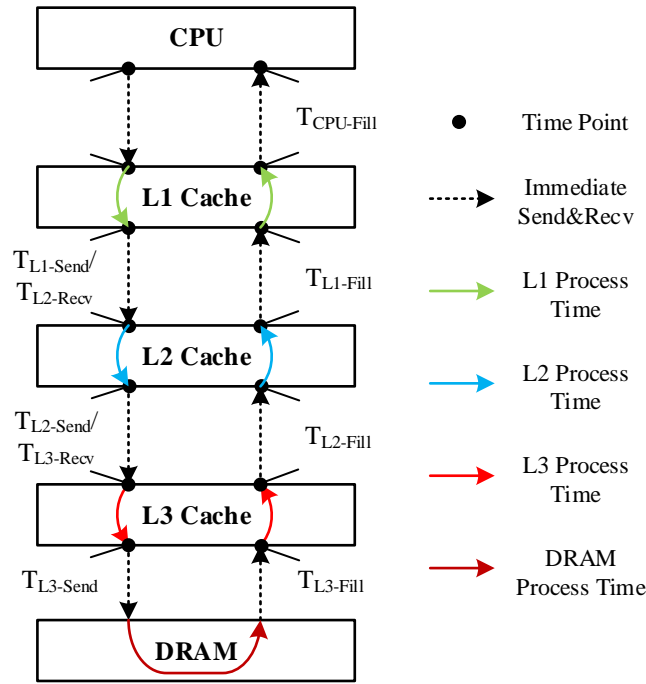


图 3.8 预取处理时间的组成

为了实现对预取的处理时间的详细统计，本文对预取处理时间的组成情况进行了详细的划分。图 3.8 展示了不同存储层级处理时间的基本组成，其中虚线箭头表示的处理过程实际上不会占用真实时间。因此上一级发送请求的时间和下一级收到请求的时间其实是相同的。对于预取请求，比如 L2Cache 生成的预取请求，在被分配 MSHR 表项时会额外设置一个时间点，模拟 L1 发送请求或者 L2 收到请求的时间点。一个 Cache 的处理时间一般来说包括两部分，一个是向下传递请求时的等待延迟，另一个是向上反馈请求数据的等待延迟。除了基本的处理延迟外，向下传递的延迟主要是请求在 MSHR 或者写缓冲区中的等待时间，向上传递的延迟则主要是请求在请求反馈队列中的等待时间。

表 3.4 给出了预取统计分析模块可以统计所有数据以及简要的说明，下面对其中标注的内容进行进一步的说明：

- a) 为了进一步确定预取多核之间的干扰，加入了处理时间统计变量。针对目标层级不同的预取，进行有效处理的存储层级并不一致。因此本文并没有使用总时间，而是使用拥有对应层级有效处理的预取的平均值作为统计结果。通过观察不同层级存储结构处理请求耗费的时间，可以更容易的确定预取效率变化的主要原因。

表 3.4 预取统计分析模块的统计数据

统计数据名称	说明
demandReqHitTotal_	不同核心的需求请求的命中次数
demandReqHitCount_	不同核心的需求请求在不同 Cache 的命中次数
demandReqMissCount_	不同核心的需求请求在不同 Cache 中的缺失次数
prefHitCount_	不同核心的不同 Cache 发出预取请求在不同 Cache 中的命中次数
prefFillCount_	不同核心的不同 Cache 发出预取请求对应数据在不同 Cache 中的填充次数
prefAvgProcessCycles_ <sup>[a]</sup>	不同核心的不同 Cache 发出预取请求在不同存储层级中的平均处理时间
prefIssuedCount_	不同核心的不同预取器生成的预取请求（不一定分配 MSHR）个数
prefAvgWaitingCycles_	不同核心的不同预取器生成的预取请求在预取请求队列中的平均等待时间
shadowedPrefCount_ <sup>[b]</sup>	不同核心的不同 Cache 发出的被需求请求覆盖的预取请求个数
squeezedPrefCount_ <sup>[c]</sup>	不同核心的不同 Cache 发出的被预取请求覆盖的预取请求个数
prefTotalUsefulValue_	不同核心的不同 Cache 发出的预取请求的核内/核间绝对有效性的总和
prefUsefulDegree_	不同核心的不同 Cache 发出的预取请求的核内/核间绝对有效性在不同范围内的个数
prefUsefulType_ <sup>[d]</sup>	不同核心的不同 Cache 发出的预取请求的不同类型预取个数
totalStatsPref_ <sup>[e]</sup>	不同核心的不同 Cache 发出并进行统计的预取总个数
dismissedLevelUpPrefNoWB_ <sup>[f]</sup>	不同核心的不同 Cache 发出的因为没有写缓冲区导致被忽略的提级预取个数
dismissedLevelUpPrefLate_ <sup>[g]</sup>	不同核心的不同 Cache 发出的因为到达太晚导致被忽略的提级预取个数
dismissedLevelDownPref_ <sup>[g]</sup>	不同核心的不同 Cache 发出的为了避免降级预取颠簸被忽略的降级预取个数

b) 该统计变量用于统计被需求请求覆盖的预取请求。在 Cache 为一个时效性较差



的预取分配 MSHR 表项后，相应的需求请求可能在该预取发出之前便命中了对应的 MSHR 表项而导致覆盖。

- c) 预取可以被需求请求覆盖，当然也可以被其他预取请求覆盖。依据后续章节提到的，预取请求在 MSHR 中的合并规则，因为合并而被无效化的预取便属于该类预取。
- d) 为了实现预取分类，每一个预取信息都会记录核内有效性 ( $\beta_{self-use}$ )、核内有害性 ( $\beta_{self-harm}$ )、核间有效性 ( $\beta_{other-use}$ ) 和核间有害性 ( $\beta_{other-harm}$ ) 的具体数值。而在数值更新上，统计模块针对多核心场景进行了适应性的调整。在多核场景中，一个需求请求可以来自于多个核心。因此在更新相关数据的时候，会基于每一个相关的核心进行更新以获得最准确的结果。比如 A 请求命中了核心 0 的预取 B 数据，而 A 请求是由核心 0 和核心 1 发出的请求合并得到的，那么该预取的核间有效性会加 1，核内有效性同时也会加 1。
- e) 由于更新周期的存在，并非所有的预取都会被统计。但由于更新周期相比于整体的运行时间很短，因此最后统计的预取数量虽然少于实际上预取器发出的有效预取请求数量，但是差值很小。同时所有统计数据都是基于该统计变量计算的，因此并不会产生影响数据分析的误差。
- f) 依据后续章节提到的提级预取的处理规则，部分提级预取会因为上一级 Cache 没有空余的写缓冲区表项，导致预取请求无法被接收而被无效化。
- g) 依据后续章节提到的提级预取的处理规则，部分提级预取会因为上一级 Cache 已经为同地址的需求请求分配了 MSHR 而产生冲突，导致预取请求无法被接收而被无效化。
- h) 依据后续章节提到的降级预取的处理规则，为了避免降级预取的反复生成和处理，本文在预取器中设置了额外的结构，对降级预取进行记录和查询。因为反复发送而被无效化的冗余降级预取，会被统计到该统计变量中。

### 3.4 数据预取有害性统计模块

经典的预取器设计中常常将预取时效性作为评价预取的标准之一。预取请求数据到达 Cache 和对应需求请求访问数据之间的时间间隔越短，那么预取的时效性就越好。之所以强调预取的时效性，是因为过早的预取会造成 Cache 污染，过晚的预取产生了功耗浪费。但是并非所有过早的预取都会产生 Cache 污染。如果一个预取替换了一个短时间内都不会再次使用的数据，同时该预取数据在被替换之前，被对应需求请求命中，那么该预取本质上并没有产生负面影响，即便该预取是一个过早的预取。因此时效性并不能完全准确的评价预取的负面效应，相对的，判断预取请求数据在 Cache 存

在期间产生的实际影响，比如被预取替换数据是否被访问，具有更准确的评价效果。为了对这种效应进行准确的统计，并在之后将结果用于数据预取过滤器模块的训练，本文添加了预取有害性统计模块。

### 3.4.1 模块结构设计

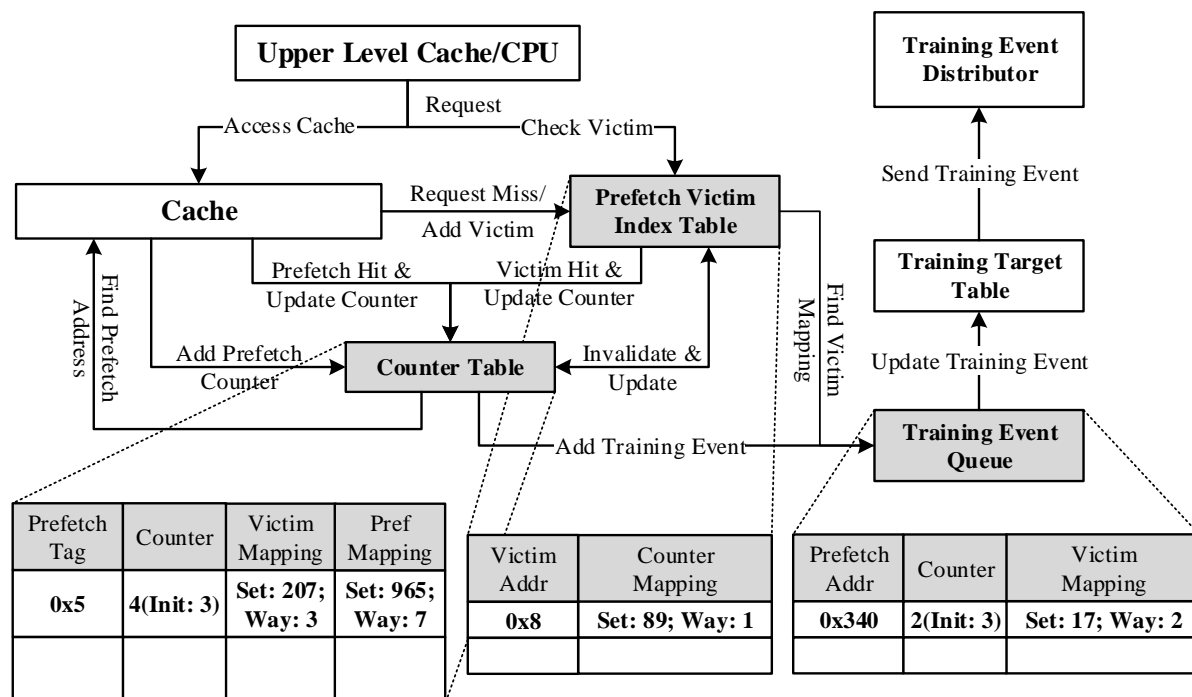


图 3.9 预取有害性统计模块的设计结构

为了实现有害性的统计，需要对预取替换数据的地址信息进行记录，这需要一些额外的存储空间进行支持。图 3.9 展示了模块的主要结构，该模块为给定的 Cache 添加了两个结构用于统计有害性信息。其中预取替换数据信息表 (Prefetch Victim Index Table) 用于存放被预取替换的数据地址信息，而预取有效性计数表 (Counter Table) 用于进行预取相关的绝对有效性计数。这两个结构都使用了组相联结构设计，并使用 LRU 替换算法。预取替换数据信息表使用被预取替换的数据地址作为索引，存放的是到预取有效性计数表的映射数据。预取有效性计数表使用预取地址作为索引，存放着一个用于绝对有效性计数的 3-bit 饱和计数器、到预取替换数据信息表的映射数据以及到 Cache 中预取缓存行的映射数据。预取有害性统计模块无需为缓存行添加任何额外的存储内容，唯一需要的预取标记位 (1bit)，在缓存行的标志位中已经存在。如果希望由 Cache 的预取索引到预取有效性计数表中，直接用 Cache 记录的 Tag 和 Set 编号合并成完整地址即可。

由于不需要记录全部的预取相关信息，两个结构的大小可以进行自定义的配置。当该结构被用于高层级 Cache，比如 L1DCache 中时，可以设计的小一些。相对的，如

果该结构被使用到 LLC 这种低层级 Cache 中时，则需要设计的大一些以便存放更多预取的信息。由于预取相关信息是唯一绑定的，每一个存在于 Cache 中的预取，只能同时和预取有效性计数表及预取替换数据信息表各自的一个表项关联。基于充足的关联关系，才可以保证预取有害性的正确统计。传统的方法会直接使用地址记录关联关系，但本文为了减少记录关联所需要的存储开销，无论是从预取替换数据信息表到预取有效性计数表的关联信息，还是预取有效性计数表到预取替换数据信息表的关联信息，都使用了 Set 和 Way 的编号记录关联关系。因此，预取有效性计数表和预取替换数据信息表除了需要支持常规的地址访问以外，还需要支持使用 Set 和 Way 编号直接关联的访问方式。当然，相对于基于地址访问的逻辑，支持这种访问方式的逻辑开销要小很多。

除了这两个结构外，有害性统计模块还增加了一个基于 FIFO 设计的训练事件队列（Training Event Queue）用于缓存预取训练事件。由于有害性统计要发挥作用，需要和后续章节说明的数据预取器控制器的训练关联。在预取信息被删除的时候，会生成一个预取训练事件，随后传送到匹配的预取过滤器模块进行训练。因为每个周期最多只能处理一个预取训练事件，因此无法得到立即处理的预取训练事件会被缓存到训练事件队列中。由于预取有害性统计模块每个周期可能生成的预取训练事件有限，训练事件队列设计的并不大。其中训练目标记录表（Training Target Table）被用来查询预取对应的训练目标，训练事件分发器（Training Event Distributor）负责训练事件的分发，这两个结构和多核场景下基于协同共享多模块设计的预取过滤器模块有关，会在后续介绍预取过滤器模块的章节做进一步的说明。

### 3.4.2 有害性的记录与更新

预取有害性统计模块需要对所有预取的相关信息进行记录和更新，其功能虽然和前面说到的预取统计分析模块十分相似，但该模块是在考虑硬件实现的基础上设计的。因此预取有害性统计模块的记录和更新，会有更多基于硬件实现上的考虑。下面会从预取信息的添加和预取信息的更新两个方面进行说明：

#### 1) 预取信息的添加

当一个新的预取请求数据填充到 Cache 中时，会进行预取信息记录的更新。在预取请求反馈访问 Cache 时，会并行访问预取有效性计数表。并尝试使用预取地址初始化一个空表项或者替换一个旧表项，表项中的计数器会被初始化为给定的数值。通过改变初始化数值，可以调节有效性或者有害性的记录上限。在预取有效性计数表初始化完成后，会同时将新表项的映射地址，即 Set 的物理编号和 Way 的物理编号，发送给预取替换数据信息表用于填充对应表项的映射地址数据。如果在初始化预取有效性计数表表项时发生了替换，那么被替换的数据会被用来索引预取替换数据信息表，并

无效化相关联的记录。替换导致删除的预取信息会被用来生成一个训练事件，对应的预取请求地址、计数器数值会被打包成为训练事件并添加到训练事件队列中，并在下一个周期发送出去。由于被替换的数据可能会被预取替换数据信息表使用，因此对应预取替换数据信息表的映射地址也会被记录到训练事件队列中，但是该地址并不会用于生成预取训练事件。

如果预取没有替换任何有效数据，而使用了空缓存行，那么添加预取信息的操作就已经完成了。如果预取替换了 Demand 类型的数据，则会直接使用替换数据的地址访问预取替换数据信息表，并选择一个空表项或者替换一个旧表项来存放数据。由于此时预取有效性计数表的分配已经完成，并已经将对应表项的映射地址发送了过来，因此可以直接使用接收到的映射地址填充新表项的数据。如果预取数据替换了预取数据，则不会使用替换的预取地址作为替换数据地址，而是继承被替换预取的替换数据地址。因此需要使用替换预取的地址访问预取有效性计数表，并对比训练事件队列最近添加的预取训练事件地址。两者至少有一个会命中，命中后则会直接使用被替换预取在预取替换数据信息表中的映射地址找到对应的表项，将该表项在预取有效性计数表中的映射地址更新为最新的地址即可。当然，在预取替换数据信息表中获取新表项的时候也会发生替换，如果发生了替换，则会利用替换表项的映射地址访问预取有效性计数表并将对应的表项无效化，生成一个新的预取训练事件填充到训练事件队列中，并尝试在下一个周期发送出去。

## 2) 预取信息的更新

当 Cache 收到访问请求时，会同时对 Cache 和预取替换数据信息表进行访问。如果在 Cache 中命中了预取数据，而请求是一个需求请求，则会使用预取地址访问预取有效性计数表，并将绝对有效性计数器递增，同时忽略来自预取替换数据信息表的有害性更新请求。如果访问 Cache 未命中，而请求是一个需求请求，同时命中了预取替换数据信息表，则会使用预取替换数据信息表表项中的映射地址访问预取有效性计数表，并对其中的绝对有效性计数器递减。如果两边均未命中，则不会进行任何操作。

因为有效性训练及无用预取的训练是不依赖于预取有害性统计模块的，预取有害性统计模块仅仅负责有害预取的训练。所以在预取有害性统计模块在生成训练事件的时候，会基于预取的绝对有效性数值进行判断，当且仅当该预取是一个有害预取时，训练事件才会被缓存到训练事件队列中，以避免冗余的训练操作。

## 3.5 数据预取过滤器模块

数据预取过滤器模块是数据预取控制器的核心模块，该模块是在 Eshan Bhatia 设计的 PPF 的基础上扩展并优化得到的。PPF 是一个基于感知器学习的预取过滤器结构，

这种学习思路最初曾由 Jiménez 引入到分支预测器中<sup>[52]</sup>，并获得了不错的性能提升。本文除了为 PPF 添加预取有害性训练支持外，还添加了多核心场景下的训练支持，预取过滤后目标层级提升的支持，丰富了 PPF 的训练场景并改进了 PPF 的训练过程。通过增加少量的存储空间，即可获得更准确的训练效果，让预取发挥更大的作用。

### 3.5.1 模块结构设计

数据预取过滤器模块拥有一个基本结构，该结构支持 Cache 间共享、CPU 间共享及全局共享。在不对数据结构共享的多核场景下，会有着不同的全局结构设计。因此本节会从单模块结构和协同共享的多模块结构两个方面介绍数据预取过滤器的结构设计。

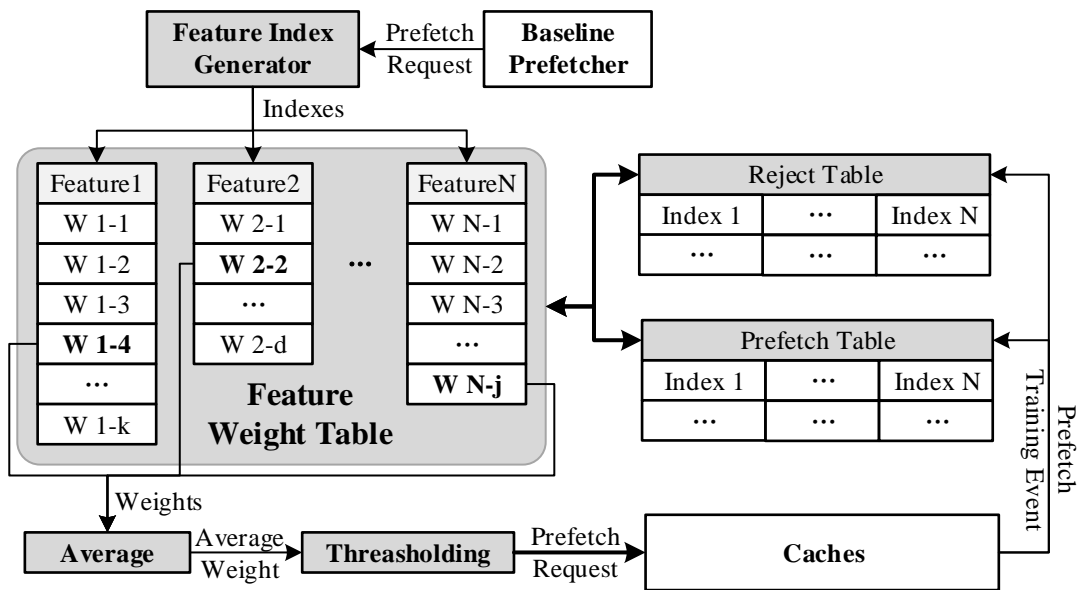


图 3.10 数据预取过滤器的单模块结构

#### 1) 单模块结构设计

图 3.10 展示了数据预取过滤器的单模块结构设计。预取过滤器的单模块结构与论文中的 PPF 设计基本一致，但是最终用于预取过滤的数值计算方法有一些调整。单模块的主要结构包括预取特征权重表（Feature Weight Table）、有效预取记录表（Prefetch Table）和过滤预取记录表（Reject Table）三个部分。

预取特征权重表是多个表格的集合体，每一个表格对应着一个特征。每一个表格均采用直接映射，即使用索引表格的数值获取 Set 找到对应的表项之后，不进行 Tag 比较直接进行读写操作。表格使用特征的数值进行索引，而特征的数值基于预取携带的相关信息，由预取特征索引生成器（Feature Index Generator）生成。表格表项中默认情况下存放着一个 5-bit 的饱和计数器，记录着一个特征数值对应的权重大小。不同特征的权重表格大小可以分别进行配置。由于没有设置 Tag，对于同一个特征，如果表格大

小变小，那么特征值的区分度就会变小，读写更新时发生冲突的可能性也会提升。

有效预取记录表和过滤预取记录表的结构十分相似，两者均为组相联结构，使用缓存行的地址进行索引，存放着一个地址对应预取所有特征的历史索引或生成特征数值所需要的元数据。有效预取记录表中存放的数据是未被过滤的预取请求对应的信息，而过滤预取记录表存放的是被过滤的预取请求对应的信息。

## 2) 多核场景下的模块结构设计

针对多核心场景，本文对 PPF 进行了训练的优化，因此多核场景下的模块设计结构会多出一些辅助结构。在多核心场景下，每一个预取器都有可以拥有一个数据预取过滤器单模块结构，如何协同好这些模块的训练和过滤是多模块结构设计中的主要挑战。

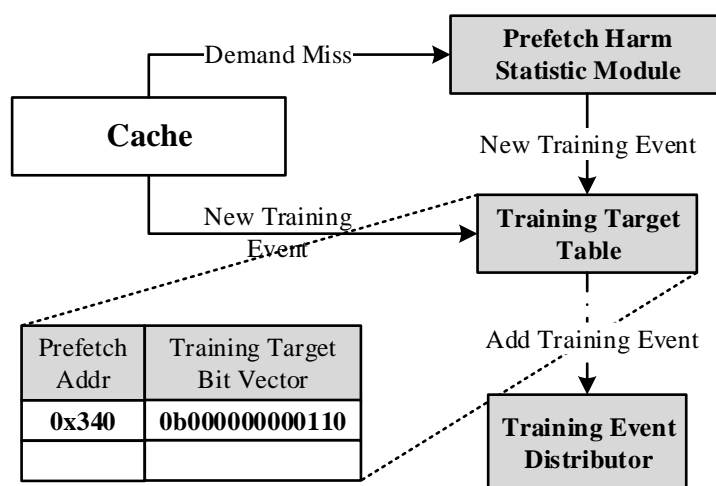


图 3.11 Cache 中生成预取训练事件的结构

本文为 PPF 添加了跨层次 Cache 训练和共享 Cache 的训练支持，设计的关键在于将 Cache 中的预取数据和所有相关的训练目标预取过滤器模块关联起来。因此训练目标关联信息需要被记录下来，然而直接在缓存行中添加数据记录产生的存储开销是不合适的。如果希望在一个 16 核心的处理器中的 LLC 中记录这些数据，同时 16 个核心的每个 Cache 层级都配有预取过滤器模块，那么记录训练目标信息需要一个 33bit 的位向量。假设每一个缓存行的数据大小约为 64 字节，对于一个 16MB 的 LLC，添加预取训练目标所需要的额外存储空间为 2MB。2MB 的空间已经远远大于 L2Cache 的容量，同时在预取数据占比较低的情况，这些额外空间的利用率也会变得很低。因此本文并没有使用这种记录方法，而是使用了如图 3.11 所示的结构设计记录训练目标信息。

该设计会为每一个 Cache 增加一个训练目标记录表结构。训练目标记录表使用组相联结构，用预取地址作为索引，存放着预取训练目标的位向量。如果当前 Cache 中存放的预取数据和某一个预取过滤器模块相关，那么对应表项中的训练目标位向量里，对应于该预取过滤器模块的位就会被设置为 1。由于不同层级的 Cache 大小并不一样，

因此可以通过配置不同大小的训练目标记录表，来保证表格的高效使用。相比于直接在缓存行中记录相关信息，使用 4096 个表项的训练目标记录表，可以将消耗的存储资源从 2MB 降低至 48KB，大大减少了存储开销。训练目标记录表不仅可以用于完善 Cache 自身产生的训练事件信息，也可以用于完善预取有害性统计模块生成的预取训练事件信息。完整预取训练事件会被发送到训练事件分发器结构中进行分发。

和预取统计分析模块一样，预取过滤器模块同样支持预取信息合并和无效化的向下传递。两者都是通过预取过滤器模块间直连的通信总线进行的。为了简化设计，如果同一个周期出现多个预取合并或者无效化请求，只有一个会得到处理。由于拥挤的情况出现频率不高，这样做不会产生较大的影响，同时还可以避免设计缓冲队列及复杂的同步逻辑。预取合并时，会直接利用预取地址访问训练目标记录表，如果命中则会对训练目标位向量进行合并，否则尝试分配一个新表项。预取无效化时直接将训练目标记录表中对应的表项无效化即可。

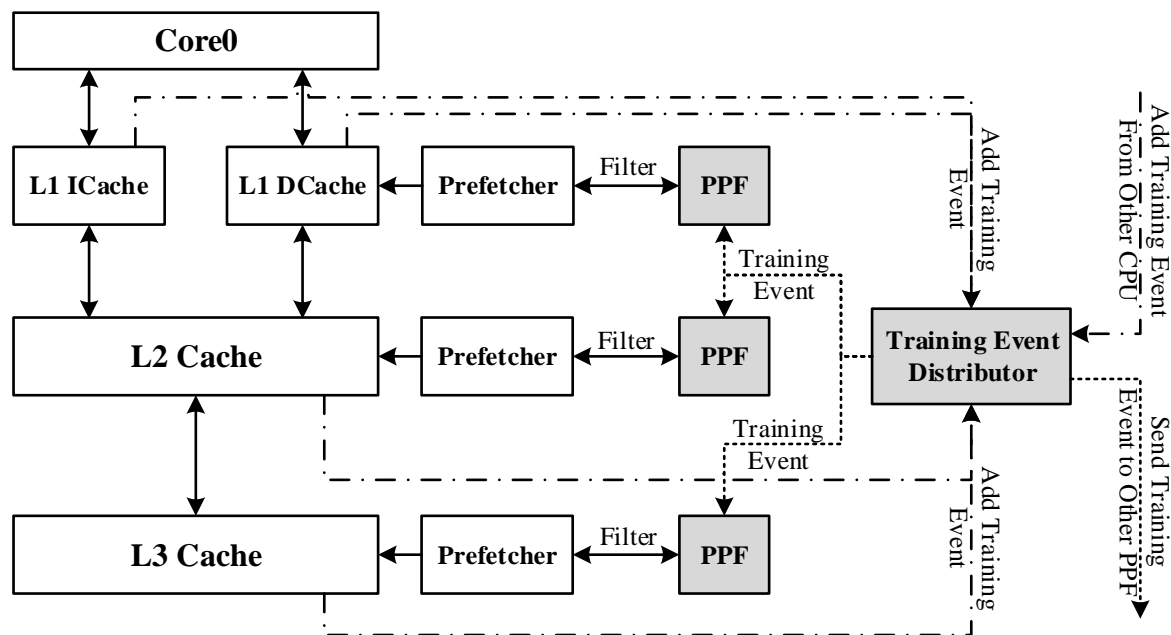


图 3.12 多模块下预取过滤器的结构设计

为了进一步节省协同共享环境下的多模块设计存储开销，单模块可以被多个结构共享。比如同一个 CPU 的私有 Cache 共享预取过滤器单模块的主要存储结构。由于 CPU 的私有 Cache 对应的数据往往来自于同样的工作集，共享存储结构并不会带来严重的训练污染。同时不同级别的预取器可以受益于相同的预取训练事件，也节省了存储开销。为了区分不同 Cache 的预取，本文也添加了用于区分不同预取器发出预取请求的特征到预取过滤器模块中。如果在多核场景下，不同核心采用多进程或者多线程的方式执行了相同的工作集，那么在不同 CPU 之间共享也是一个选择。但因为这样的场景出现并不频繁，CPU 间共享并未被采用。

图 3.12 展示了本文基于多核场景下重新设计的预取过滤器结构，预取训练请求可能来自于不同的 Cache，而同一个预取数据生成的训练事件可能会对多个预取过滤器模块有效。因此本文设计了训练事件分发器，负责预取训练事件的处理和分发。图 3.13 展示了训练事件分发器的内部结构设计，它会从不同的 Cache 中接收预取训练事件，并基于给定的训练目标位向量定位到目标预取过滤器模块，将预取训练事件发送给对应的预取过滤器模块。由于一个预取过滤器模块每一个周期同时只能执行一个训练事件，训练事件分发器会为每一个产生预取训练事件的 Cache 设置一个只有 2 个表项的 FIFO 队列，缓存无法在当前周期进行处理的预取训练事件。

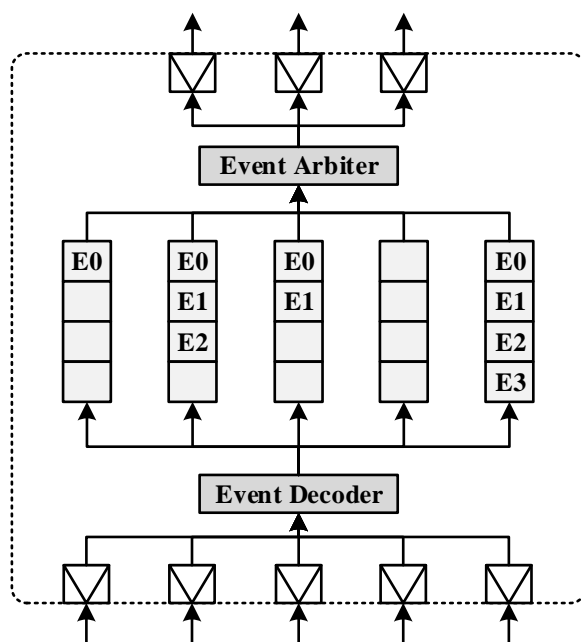


图 3.13 训练事件分发器的结构

训练事件分发器中的队列处理策略与预取请求队列的设计十分相似，新的预取训练事件会直接将最老的训练事件挤掉。因此队列大小的限制会导致一部分预取训练事件的无效化。由于接收的预取训练事件可能是多目标的，同一个周期可能会有多个预取训练事件同时尝试插入同一个队列中，这时候会随机选择一个训练事件插入到队列中。由于同一个周期可能有多个，针对相同目标预取过滤器模块的训练事件准备就绪，本文还为训练事件分发器配置了预取训练事件仲裁器（Event Arbiter）对训练事件进行仲裁。默认配置下仲裁器会优先选择训练幅度最大的训练事件，发送到目标预取过滤器模块进行训练处理。

### 3.5.2 预取目标层级的变更

Eshan Bhatia 设计的 PPF 支持预取目标层级的降低，然而 Gem5 模拟器本身并不支



持对预取目标层级的调整，因此本文为 Gem5 添加了预取目标层级的降低的支持。预取过滤的思路常常是对不好的预取进行惩罚，却没有相应的奖励机制，因此本文还向预取过滤器模块中添加了预取目标层级提升的支持。在预取目标层级变更的支持下，单一 Cache 层级配置的数据预取器便可以将预取数据的覆盖范围扩展到所有 Cache 层级，这对低功耗处理器来说是十分有益的尝试。

同时，L1Cache 看到的访问地址并非缓存行对齐的，实际训练出的步长常常小于缓存行的大小，然而预取是以缓存行为单位生成的，这会使得 L1Cache 的预取器效率变低。相对的，L2Cache 看到的所有访问地址都是缓存行对齐的，对应数据预取器训练出的步长一定是缓存行大小的整数倍，因此具有更高的效率。而传统的数据预取器设计不允许 L2Cache 的预取器将准确的预取数据放到 L1Cache 中，使得预取效率受到了限制。本文添加的预取目标层级提升支持，便打破了这个限制，让单个预取器数据实现更加全面的覆盖和发挥出更加出色的预取效果。

从实现的角度，预取目标层级的变更主要有两个策略：其中一个是将调整过目标 Cache 层级的预取发送到目标 Cache 的预取请求队列中；另一种则是将目标层级信息记录在传递的请求中，请求会依据记录将数据存放到目标层级 Cache 中。前一种方法由于没有更改请求信息，在请求的处理过程上无需调整，但是不同层级 Cache 之间预取请求的传递依旧会产生严重的 IO 开销。因此本文选择了后一种方法实现，虽然在请求处理上有一定的调整，但是不需要增加额外的硬件支持不同 Cache 之间预取请求的传递。

下面会从预取目标层级的降低、预取目标层级的提升、MSHR 的请求合并三个方面介绍本文在预取目标层级变更上的工作：

### 1) 降低预取目标层级

如果预取的目标层级被降低，预取请求依旧可以正确的分配 MSHR 表项。但是在准备就绪并生成请求发送成功后，MSHR 表项便会被立即释放。在数据不能达到的 Cache 层级，对应的 MSHR 表项都会在生成请求并成功发送后释放。在目标 Cache 层级以及目标层级以下的 Cache 中，则会按照正常的预取流程处理。在数据反馈过程中，在数据没有到达目标 Cache 层级前都会进行数据的填充。当数据到达目标 Cache 层级后，将不会继续向上执行请求反馈，至此一个目标 Cache 层级被降低的预取请求完成所有的处理流程。

由于 MSHR 表项在生成请求后就会被释放，会出现图 3.14 这样一组降级预取反复生成并发送的情况。对于一个 PC 对应的准确跨步预取，必定会在每次执行访问时触发新一轮的预取。如果预取深度大于 1，那么前后两次预取必定存在重复的预取请求。由于预取器在将预取请求插入到预取请求缓冲队列之前，会经由预取过滤器进行冗余消除，这些预取请求常常会因为预取请求缓冲队列中已经存在、命中了 MSHR 或者命

中了当前 Cache 数据而被过滤。但如果这些预取被数据过滤器识别并认定为需要降低目标 Cache 层级的预取，同时当前的访问请求不密集，MSHR 表项很快就可以准备就绪并得到处理。那么这些重复的预取请求就会被反复插入到预取缓冲队列中，并分配 MSHR 和发送。这样不仅浪费了 Cache 的带宽，还产生了额外的功耗开销。

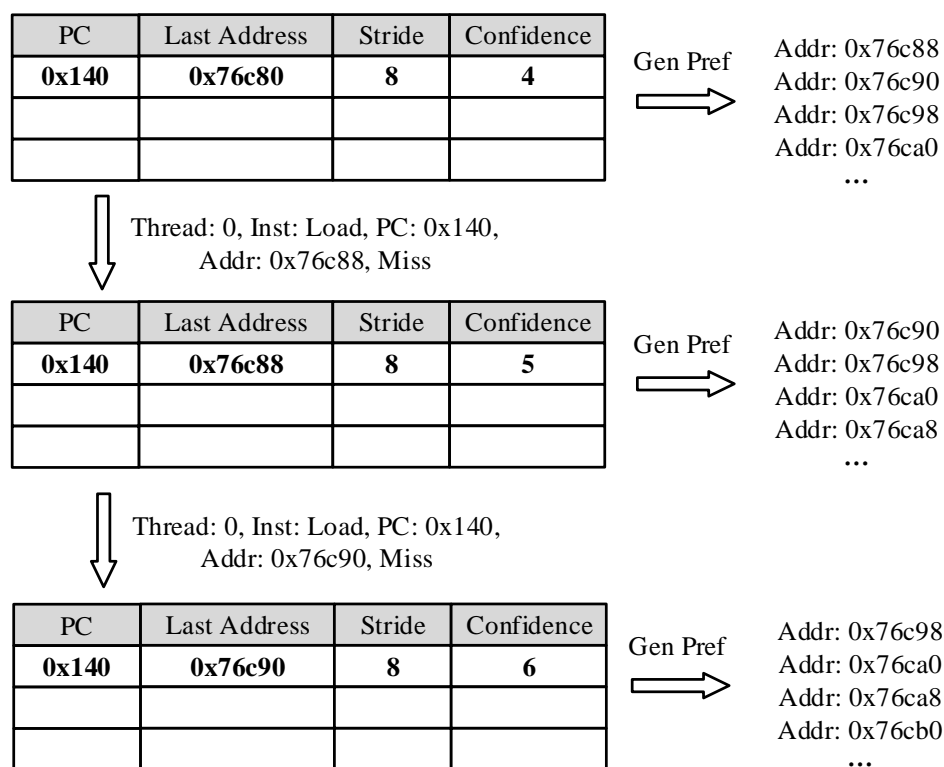


图 3.14 基于固定步长预取器中的重复预取

为了消除冗余的降级预取请求，本文为预取器中的预取过滤器添加了一个降级预取请求记录缓冲区，存放最近发送降级预取地址。这个降级预取记录缓冲区大小一般设置为预取器的最大预取深度。每一次预取器生成新的预取时，都会对降级预取记录缓冲区进行批量更新。在预取过滤器过滤降级预取请求的时候，除了进行冗余检查，还会对降级预取请求缓冲区进行检查，如果发生命中，则降级预取也会被过滤掉以避免带宽和功耗的浪费。

## 2) 提升预取目标层级

相比于被降低目标 Cache 层级的预取请求，提升预取目标 Cache 层级的预取请求处理流程十分相似。主要的区别在于请求从 MSHR 表项生成并发送出去后，对应的 MSHR 表项并不会被释放。当预取的请求反馈到达产生预取请求的 Cache 层级后，会处理对应的 MSHR 表项并释放。由于预取的目标 Cache 层级得到了提升，当前 Cache 在释放 MSHR 表项后会继续向上传递请求反馈。而更高层次的 Cache 在接收到请求反馈后，会进行数据替换和填充，新填充的缓存行会设置预取属性标记位。如果处理请求

反馈的 Cache 发现预取请求的目标 Cache 层级正是当前层级，则停止向上传递请求反馈。至此，一个预取目标 Cache 层级被提升的预取请求处理完毕。

理想的情况下，提级预取在生成预取请求 Cache 的上层 Cache 中都能得到正确的处理。然而有时预取请求反馈的处理可能并不及时，比如上层 Cache 已经收到了对应的需求请求并分配了 MSHR 表项，或者上层 Cache 中已经存放了预取对应的数据，那么提级预取将会被忽略。此外，预取数据的填充可能会导致数据的替换，而替换数据的写回操作可能会占用写缓冲区的表项。因此在上传提级预取请求的过程中，如果发现上层 Cache 中没有足够的写缓冲区表项存放预取数据替换所产生的写回请求，则提级预取也会被忽略。实际上在 Cache 中，写缓冲区会为处理中的请求保留一定数量的空表项，以保证这些请求反馈到达并进行数据替换和填充时，不会因为缺少写缓冲区表项存放替换数据产生的写回请求而出现错误。针对提级预取计算可用写缓冲区表项数目会减去保留的空表项，以保证处理中的需求请求不会因为提级预取占用写缓冲区表项而产生处理错误。

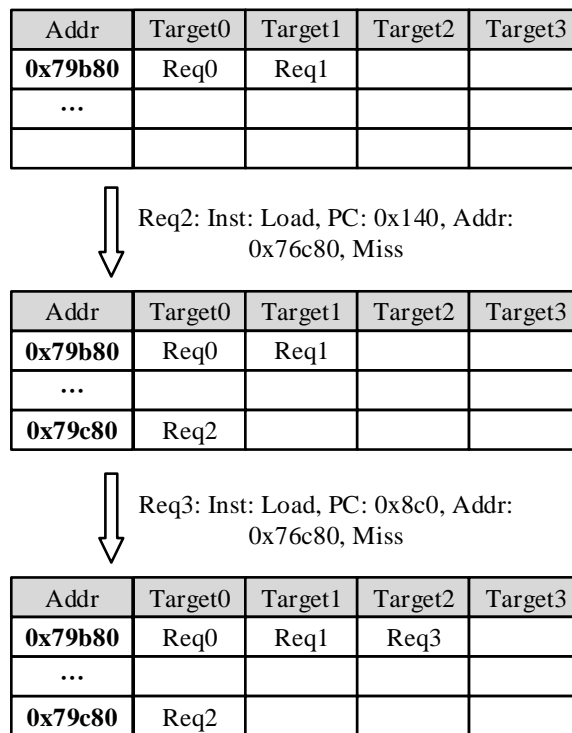


图 3.15 Gem5 中 MSHR 的工作机制

### 3) MSHR 中的请求合并

图 3.15 展示了 Gem5 中 MSHR 的工作机制，MSHR 的每一个表项对应一个缓存行地址。单个表项内还允许存放多个请求信息，以便于请求的合并，减少向下传递的访问请求个数。因此无论是 MSHR 表项全满，还是任意一个 MSHR 表项中的目标列表被填满都会导致 Cache 的阻塞。

由于本文支持了预取目标 Cache 层级的变更，预取 MSHR 中出现了更多复杂的场景。因此本文针对不同的请求合并场景，也设计了合适的策略以保证请求的正确传递和处理。在非共享 Cache 的预取 MSHR 中，不会出现来自不同核心的请求，因此无需处理不同核心之间的请求冲突。在共享 Cache 的预取 MSHR 合并新请求时，则可能会对其他核心的请求进行特殊的处理。下面会分别从这两个方面对 MSHR 中请求合并的处理方法进行说明。

$$Pref_{local}^{high/local/low} ? Pref_{high}^{local/low} * (Demand_{high}^{high} | Pref_{high}^{high}) ? \quad (3-3)$$

在非共享 Cache 的预取 MSHR 表项中，即单核场景下可能出现请求合并方式可以由式（3-3）表示。该公式按照正则表达式的形式表明了可能的请求合并序列。其中  $Type_{source}^{target}$  表示一个类型为 Type 的请求，Pref 表示预取，Demand 表示需求请求。产生这个请求的 Cache 层级在下标表示，目标 Cache 在上标表示。在层级表示上，local 对应于 MSHR 所在的 Cache 层级，high 对应于更高 Cache 层级，low 对应于更低的 Cache 层级。 $Pref_{local}^{high/local/low}$  是来自 MSHR 所在 Cache 的请求，由于预取器中 Prefetch Filter 对预取的过滤效果，此类请求至多出现一个。 $Pref_{high}^{local/low}$  不会在高层级 Cache 中保留 MSHR 记录，因此可以不受限制的向下发送并合并到 MSHR 表项中。由于  $Demand_{high}^{high}$  和  $Pref_{high}^{high}$  均会在高层级 Cache 中保留 MSHR 记录，所以这两类请求最多只会合并一个。而同时连接到 L1DCache 和 L1ICache 的 L2Cache，属于共享 Cache 的一种特殊情况，并不在式（3-3）表示的范围内。

表 3.5 非共享 Cache 中预取 MSHR 的请求合并规则

MSHR 状态	需要合并的请求类型	合并操作
未处理	$Pref_{local}^{high/local/low}$	不会合并，必定分配新的 MSHR 表项
	$Pref_{high}^{local/low}$	选择最高层级最新的预取决策，无效化之前有效的预取（ $Pref_{local}^{high/local/low}$ 或 $Pref_{high}^{local/low}$ ）
	$Demand_{high}^{high}   Pref_{high}^{high}$	将新的请求作为有效的请求，无效化之前有效的预取（ $Pref_{local}^{high/local/low}$ 或 $Pref_{high}^{local/low}$ ）
处理中	$Pref_{local}^{high/local/low}$	不可能的情况
	$Pref_{high}^{local/low}$	新的请求将会被无效化
	$Demand_{high}^{high}   Pref_{high}^{high}$	新的请求不会被无效化，并且在请求数据到达时能够得到正确的反馈

在非共享 Cache 的预取 MSHR 表项中，为了更高效地利用 MSHR 表项的空间，同时只会有一个预取请求生效。相应的请求合并规则在表 3.5 中展示了出来，请求合并不仅会在 MSHR 表项对应的请求发送前出现，还可能出现在 MSHR 的请求向下发送之

后。如果请求合并到的 MSHR 处于处理中状态，那么贸然无效化已经合并的请求会导致请求反馈无法得到正确的响应，因此合并规则会有所不同。

表 3.6 共享 Cache 中预取 MSHR 的请求合并规则

MSHR 状态	需要合并的请求类型	合并操作
未处理	$Pref_{local}^{high/local/low}$	不会合并，必定分配新的 MSHR 表项
	$Pref_{high}^{local/low}$	选择最高层级最新的同源预取决策，无效化 $Pref_{local}^{high/local/low}$ 和同源 $Pref_{high}^{local/low}$
	$Pref_{high}^{high}$	选择预取请求中提升目标层级最高的预取决策，无效化 $Pref_{local}^{high/local/low}$ 以及所有 $Pref_{high}^{local/low}$
	$Demand_{high}^{high}$	将新的请求作为有效的请求，无效化 $Pref_{local}^{high/local/low}$ 以及所有 $Pref_{high}^{local/low}$
处理中	$Pref_{local}^{high/local/low}$	不可能的情况
	$Pref_{high}^{local/low}$	新的请求将会被无效化
	$Demand_{high}^{high}   Pref_{high}^{high}$	新的请求不会被无效化，并且在请求数据到达时能够得到正确的反馈

共享 Cache 中预取 MSHR 表项的请求合并规则，是在非共享 Cache 的预取 MSHR 表项请求合并规则的基础上扩展出来的。针对同一个上级 Cache 发送的请求，请求合并的规则保持不变，但是新的请求合并可能会导致其他上级 Cache 的请求被无效化。同时由于预取请求可以来自于多个不同的上层 Cache， $Pref_{high}^{high}$  可以同时存在多个（但是不会超过上层 Cache 的数目）。因此在共享 Cache 的预取 MSHR 表项中，会选择非同源预取请求中提升目标层级最高的预取请求作为唯一有效的预取请求，来生成向下传递的请求。

### 3.5.3 预取权重训练

预取特征权重表是数据预取过滤器模块的核心结构，存放着每一个特征数值对应的权重。如何选择预取特征是 Eshan Bhatia 论文中 PPF 的设计核心，而本文并没有将特征的选用作为研究的核心内容。但由于本文将数据预取过滤器结构扩展到了多核场景下的多层级 Cache 中，需要考虑的问题变得多样化。所以本文在针对权重的处理上进行了适应性的调整，在特征上大部分延用了原论文中的选择。表 3.7 给出了本文预取过滤器模块设计中所支持的特征，其中由本文添加的新特征被标记为了斜体。新添加的三个特征中，第一个可以更直接获取到分支信息，第二个和第三个则是为了处理在共享数据预取过滤器模块时，区分不同预取器发出预取请求的问题。

表 3.7 数据预取过滤器模块使用的特征

预取特征 (Feature)	说明	效用
Page Address XOR Confidence	触发预取访问的物理页号和可信度的异或	用于调控页表及可信度的影响，可以分辨不同页表可信度的有效性
CacheLine Address	触发预取的访问缓存行地址	关联到由同一个访问触发的整个预取流，但是只取用了缓存行的地址，去除了缓存行内偏移
Page Address	触发预取的访问地址的物理页号	关联到由同一个访问触发的整个预取流，但是只取用了页号部分，去除了页内偏移
Physical Address	触发预取的访问的物理地址	关联到由同一个访问触发的整个预取流，使用物理地址的低位
Confidence	预取器中评估的预取可信度	该可信度被用于基准预取器的预取过滤机制
$PC_1 \text{ XOR } PC_2 \gg 1 \text{ XOR } PC_3 \gg 2$	触发预取的访问指令 PC 以及之前的几个指令 PC 的哈希值	希望借助触发访问的 PC 抓取分支信息，通过移位避免高位重复数据干扰
Signature XOR Delta	签名与当前预取间隔的异或结果	签名只和 SPP 预取器有关，本文不会使用到该 Feature
PC XOR Depth	触发预取访问的指令 PC 和预取深度的异或	PC 和预取深度的结合，可以反映出一个预取流的某个深度的可信情况
PC XOR Delta	触发预取访问的指令 PC 和预取间隔的异或	PC 和预取间隔的结合，可以反映出一个访问触发预取时其间隔的可信情况
$BPC_1 \text{ XOR } BPC_2 \gg 1 \text{ XOR } BPC_3 \gg 2$	触发预取时近三次分支指令 PC 的哈希值	直接抓取分支信息，通过移位避免高位重复数据干扰
<i>PrefetcherID XOR PC</i>	预取器和触发预取的访问指令 PC 的异或	区分不同预取器对不同预取流的可信情况
<i>PrefetcherID XOR PageAddress</i>	预取器和预取所在页面物理页号的异或	区分不同预取器在不同页面中预取的可信情况

在训练更新上，Eshan Bhatia 论文中的 PPF 仅仅支持所在 Cache 层级的训练更新。而本文由于添加了训练目标记录表，使得多核场景下，不同层次 Cache 之间均可以进行训练事件的传递，大大丰富了预取权重训练的场景。加上本文对预取目标层级变更进行了扩展，不同 Cache 层级训练事件的互通也让目标层级变更的预取训练成为可能。表 3.8 展示了本文设计的数据预取过滤器模块支持的所有训练事件，其中 BadPref 需要预取有害性模块的支持。

表 3.8 预取过滤器模块的训练事件分布

训练类型	产生训练事件的场景
GoodPref	需求请求命中尚未无效化的预取数据
	需求请求合并到了处理中的预取 MSHR 中
UselessPref	降级预取命中了目标 Cache 层级的数据
	预取请求在合并到 MSHR 过程中被无效化
BadPref	预取数据被替换时预取标志位未被清除，且没有被无效化
	预取数据被替换时预取有害性统计模块认定为有害预取
DemandMiss	预取被预取有害性统计模块无效化时被认定为有害预取
	预取器所在层级的 Cache 发生了需求请求访问缺失，并且需求请求合并到了非预取 MSHR 中时

在执行训练事件的时候，预取过滤器模块都会使用训练事件中的地址查询有效预取记录表和过滤预取记录表。如果访问命中，就会使用表项中存储的不同特征的索引（如果存放的是元数据，则会首先使用预取特征索引生成器生成特征索引），对预取特征权重表中的权重进行更新。针对不同的训练类型，会有不同的更新方式，在 Eshan Bhatia 论文中的 PPF 设计中，并没有强调权重更新方式的影响，所有训练对权重的更改大小均为 1。而本文将权重更新数值作为了一个可配置参数，由于引进了不同 Cache 之间训练事件的传递，权重更新方式也需要进一步的区分。

表 3.9 预取过滤器模块的训练配置

训练类型	训练事件来源	权重更新数值
GoodPref	L1Cache	+1
	L2Cache	+2
	L3Cache	+3
UselessPref	任何	-1
BadPref	L1Cache	-3
	L2Cache	-2
	L3Cache	-1
DemandMiss	L1Cache	+3
	L2Cache	+2
	L3Cache	+1

表 3.9 给出了数据预取过滤器实现中，对不同训练类型使用的默认权重更新大小。由于需求请求主要来自于处理器核心，所以在低层级的预取被命中后，应该被更快地

提升到离处理器核心更近的 Cache 层级。因此针对有用预取,即 GoodPref 的训练类型,本文为较低的 Cache 层级生成的训练事件设置了更大的训练幅度。无用预取本身占用了 Cache 的空间,但是因为替换的数据未曾被访问,自身便被替换,因此产生的损害并不大,本文并未对 UselessPref 训练类型进行详细的划分。相对的, BadPref 是借助有害性统计模块生成的训练事件类型。既然被标记为有害预取,说明该预取被替换数据造成的性能损失要大于预取产生的性能提升,因此需要对权重进行负向训练更新。另一方面,高层级的 Cache 容量更小,实际产生的有害性会更大,因此本文为高层级 Cache 中的有害预取训练事件设置了更大的训练幅度。

如果仅仅使用预取相关的训练事件训练预取过滤器,那么当一组预取经过训练变成被过滤的预取后,由于预取不会被发送,将不会再有对应的训练将它们重新训练成为有效的预取。因此训练类型中还设置了 DemandMiss,在数据预取器所属 Cache 接收到需求请求并且发生访问缺失,同时需求请求的地址在过滤预取记录表中存在记录,则训练相关的预取,让这些预取重新起效。借助 DemandMiss 类型的训练,可以在重新进入某一段相同代码时唤醒一组相关的预取请求。为了加快唤醒预取的过程,本文为高层级 Cache 中的 Demand Miss 训练类型设置了更大的训练幅度。

### 3.5.4 预取过滤

数据预取过滤器模块中最为关键的步骤便是预取过滤。在 Eshan Bhatia 论文中 PPF 设计里,每一次过滤使用的是预取所有特征权重地加和作为过滤指标。而本文将数据预取过滤器的适用范围扩展到了多核场景下的多层次 Cache 中,因此对预取过滤过程也进行了调整。

表 1.2 展示了一些主流商用处理器或微架构中的预取策略,虽然使用不同的 ISA,但是在 AMD 和 Intel 的处理器或微架构中,均使用了一种以上的数据预取器。而表 3.7 中列出的特征并不适用于所有的数据预取器,那么直接使用权重加和作为预取过滤的标准一定会产生错误的结果。为了能让预取过滤同时适用于不同种类的数据预取器,本文使用了不同权重的平均值而非权重加和作为预取过滤的指标。由于不同的数据预取器支持的特征不同,在进行预取过滤时,每一个可能无效的特征都会设置额外的 1bit 表示当前特征是否有效。同样的,在有效预取记录表和过滤预取记录表也会为可能无效的特征索引设置额外的 1bit 作为有效位。

取平均值相比于权重加和产生的延迟要高很多,很难保证在预取生成后尽快完成预取过滤操作,势必会导致预取延迟增高。为了减少计算延迟,实际设计时会使用加和和移位操作,结合过滤阈值的变换实现。以 6 个有效特征 A、B、C、D、E 和 F 为例,在计算出所有特征权重的加和之后,会通过移位而非整数除法计算平均值。移位数等于比特征数量小的 2 的  $n$  次幂中最大值的  $n$ ,这里的移位数等于 2。由于平均值实际上



被放大了，对应的阈值也会被放大相同的倍数（6/4=1.5）。通过比较权重加和右移 2 位的结果和放大 1.5 倍后的阈值即可确定预取的目标层级。

$$\sum_{i=1}^{n_{features}} Weight[i] \gg \left\lfloor \log_2(n_{features}) \right\rfloor \stackrel{compare}{\Leftrightarrow} \frac{threshold_{level}}{n_{features}} \ll \left\lfloor \log_2(n_{features}) \right\rfloor \quad (3-4)$$

式（3-4）给出了改进后的平均值计算过程公式，其中  $n_{features}$  表示特征的总数， $threshold_{level}$  表示 level 对应 Cache 层级的阈值。表达式对应的算法中选择对平均值放大而非缩小可以避免因为缩小失去精度。为了支持这种处理，放大后的阈值及移位数会被固定到 ROM 中，可以使用生成预取请求的预取器 ID 进行访问并选择。而阈值放大的倍数不会超过 2，因此只需要额外 1bit 即可实现上述快速平均值计算，整体延迟相比于原本加和只会多出一个移位操作的延迟。

除了支持不同表格大小和训练幅度的配置，本文还可以对阈值进行配置。默认情况下不同层级阈值之间的间隔并不一致。本文使用的是 5bit 权重，针对 L1Cache、L2Cache 和 L3Cache 配置的阈值分别为 24、16、4。借助这种不均匀的阈值分布，可以同时提升预取被提升到最高 Cache 层级或者被过滤的难度。即只有预取十分有益才可能被提升到 L1Cache，十分有害时才会被过滤。选择配置不同，必定会产生不同的过滤效果，针对不同工作集也会存在不同的最佳阈值配置。

如果预取请求没有被过滤掉，那么该预取生成特征使用元数据或者特征值索引会被存放到有效预取记录表中。如果预取被过滤掉，那么该预取生成特征使用元数据或者特征值索引会被存放到过滤预取记录表中。同一个预取同时只能存在于有效预取记录表和过滤预取记录表中的一个表里面，以避免错误或者冗余的预取训练。

### 3.6 数据预取控制器的存储开销

本节会对数据预取控制器的存储开销进行分析，其中主要的两个部分分别是预取有害性统计模块和数据预取过滤器模块。下面会以表格的形式给出不同模块的存储开销。由于不同层级 Cache 中的模块大小配置不一，所以所有与所在 Cache 层级相关的元数据大小配置被标记为斜体。表格中给出是将模块使用到 L2Cache 中时的默认配置情况。

表 3.10 给出了单个预取有害性统计模块的开销，其中预取替换数据信息表和预取有效性计数表的大小会依据所属 Cache 的大小变化而变化。三个主要结构中的映射地址大小也会随着两个表的大小变化而变化。本文并没有在预取替换数据信息表和预取有效性计数表中使用完整的 Tag，借助该方法，预取有害性统计模块的存储开销降低了约 63%。

表 3.10 预取有害性统计模块的存储开销

结构	表大小	表项存储开销 (单位: bit)	说明
预取替换数据 信息表	1024	6	Tag
		1	Valid
		10	Counter Table Mapping
预取有效性计数表	1024	6	Tag
		1	Valid
		3	Saturated Counter
		10	Victim Mapping
		12	Prefetch Mapping
训练事件队列	4	16	Prefetch Address
		3	Counter Value
		10	Victim Mapping
Total		6 286Bytes (6.14KB)	

在预取有害性表格中使用不完整 Tag 时, 相应的处理策略也会有一定的变化。为了保证预取有效性计数器、预取替换数据信息和预取信息之间的唯一关联性, 预取有效性计数表中加入了到 Cache 中预取数据的映射地址。在新增预取信息记录发生替换时, 会使用该映射地址进行比对确保预取信息的完全匹配。在需要生成训练事件时, 借助该映射地址反向查询 Cache 获得预取数据的完整地址。相比于记录预取的完整地址, 记录映射地址所产生的存储开销要小很多。同时由于训练目标记录表没有使用完整的 Tag 进行匹配, 训练事件队列中也仅仅记录了预取地址中的低 16bit。

表 3.11 给出了对预取过滤器模块进行训练所需要元数据大小。按照图 3.10 的设计结构, 有效预取记录表和过滤预取记录表中存放的是不同特征的索引, 然是存放索引的存储开销可能比直接存放用于生成特征索引的元数据的存储开销更大。比如在本文的默认配置下, 按照特征索引存放一个预取信息需要 89bit, 而存放元数据则只需要 70bit。如果数据预取过滤器模块使用的特征不同, 两中处理方法的开销大小关系也会有所变化。

表 3.12 展示了单个预取过滤器模块在 8 核心场景下所需要的存储开销。有效预取记录表和过滤预取记录表的大小会依据所属 Cache 的大小而变化, 训练事件分发器的大小则和当前处理器的核心数相关。其中 Training Target 的编码长度和系统中配置的预取过滤器模块数目相关。无论是在有效预取记录表、过滤预取记录表还是训练目标记录表中, 均使用了不完整的 Tag 进行匹配, 以减少存储开销。由于有效预取记录表和

过滤预取记录表没有使用完整的 Tag 匹配, 因此训练事件中的预取地址也从 64bit 压缩到了 16bit, 这样可以有效节省训练事件的分发产生的通信设计开销。整体上看, 本文添加多核场景下的支持, 相比于原论文中单核场景下的设计, 存储开销仅仅增加了 5.7%。

表 3.11 数据预取控制器用于训练的元数据

元数据名称	元数据大小 (单位: bit)	说明
Address	12	触发预取请求的访问地址的低 12bit
PageAddress	10	触发预取请求的访问物理页号低 10bit
Confidence	3	预取请求的可信度
$PC \wedge PC_1 \gg 1 \wedge PC_2 \gg 2$	12	最近触发预取的访问 PC 哈希数值
$BPC \wedge BPC_1 \gg 1 \wedge BPC_2 \gg 2$	12	触发预取时最近分支 PC 的哈希数值
PC	12	触发预取请求的访问 PC
Depth	4	预取请求的深度
Delta	5	预取请求的固定步长
Total		70 bit

### 3.7 本章小结

本章节除了对本文主要的实验平台——Gem5 的 Cache 结构进行简要分析以外, 还对数据预取控制器中的三个主要结构, 预取统计分析模块、预取有害性统计模块和数据预取过滤器模块的设计进行了介绍。

预取统计分析模块通过将预取分类扩展到多核场景下, 实现了对多核心预取的详细分类和统计分析策略。借助预取统计分析模块, 可以更清晰地理解预取对处理器性能产生影响的原因; 预取有害性统计模块通过记录预取替换地址信息及收集请求访问的响应情况, 给出了预取绝对有害性的统计结果, 可以辅助数据预取过滤器模块的预取训练; 数据预取过滤器模块是数据预取控制器的核心模块, 本文不仅为它配置了预取有害性统计模块, 还添加了多核心场景下的训练支持, 扩展了预取目标 Cache 层级, 丰富了训练场景并改进了训练过程。

本章节在最后还对数据预取控制器的存储开销进行了分析。分析结果显示, 在 L2Cache 中使用默认配置的预取控制器结构时, 预取有害性统计模块以及对数据预取过滤器模块的改进所产生的存储开销, 仅占 Cache 本体的 3.12%, 占原数据预取过滤器存储开销的 24.35%。

表 3.12 数据预取过滤器模块的存储开销

结构	表大小	表项存储开销 (单位: bit)	说明
有效预取 记录表	1024	6	Tag
		1	Valid
		70	Metadata for Features
过滤预取 记录表	1024	6	Tag
		1	Valid
		70	Metadata for Features
预取特征 权重表	5*4096	5	Feature: Physical Address, PC ^ PC <sub>1</sub> >>1 ^ PC <sub>2</sub> >>2, PC ^ Depth, PC ^ Delta, BPC ^ BPC <sub>1</sub> >>1 ^ BPC <sub>2</sub> >>2
	2*1024	5	Feature: PageAddress, PageAddress ^ Confidence
	64	5	Feature: CacheLine Address
	8	5	Feature: Confidence
训练目标 记录表	1024	6	Tag
		1	Valid
		16	Training Target
训练事件 分发器	3*2	16	Prefetch Address
		2	Training Type
Total		36 794 Bytes (35.93KB)	

## 第四章 数据预取控制器的评测与分析

本章会对多核环境下预取之间产生干扰的原因进行分析，并对面向多核的数据预取控制器的性能进行评测。主要内容包括了对评测环境的介绍，对多核预取干扰原因的分析，在单核场景下和多核场景下对数据预取控制器地性能评测与分析，以及对数据预取控制器不同配置下性能的比较与分析。

### 4.1 评测环境

本节会分别从评测平台的配置矫正、评测程序以及评测硬件环境三个方面对评测环境进行介绍。

#### 4.1.1 评测平台的校正

表 4.1 评测使用的默认 Cache 配置

配置参数说明	参数配置
L1 ICache	64KB, 4-way, 4 Cycles, 4 MSHRs (20 Targets)
L1 DCache	32KB, 8-way, 4-5 Cycles, 4 MSHRs (20 Targets), 8 写缓冲区 s
L2Cache (Private)	256KB, 8-way, 12-18 Cycles, 20 MSHRs (12 Targets) 8 写缓冲区 s
Stride Prefetcher (L2Cache)	64 PCTable (4-way), 32 Prefetch Reuquest Queue, Degree 16
L3Cache (Shared)	16MB, 16-way、37-43 Cycles, 32 MSHRs (24Targets) 16 写缓冲区 s
DRAM Latency	170 Cycles
DRAM Channels	2
DRAM Size	8GB
DRAM Type	DDR4-2400-8*8
DRAM Ranks	8

Gem5 是一款以事件驱动的，模块化的结构级模拟器，它有机结合了密歇根大学的 m5 和威斯康星大学开发的 GEMS。集成了 ARM、ALPHA、MIPS、X86、RISCV 等多种指令系统模型，能够在 ALPHA、X86、ARM 系统上加载操作系统，而本文选择的是使用最为广泛的 X86 作为测试使用的 ISA。Gem5 还支持 Atomic、Timing、Minor 和 O3（Out of order）四种可以热插拔的 CPU 模型。其系统模型有两种，分别为系统调用和全系统模拟，本文的相关实验会选择 SE 模式进行模拟与测试。为了确保实验结果贴近于实际处理器的运行场景，本文基于 AMD 的第一代 Zen 架构对大部分 Gem5 的配置进行了校正。

表 4.1 给出了本文实验中 Gem5 的 Cache 部分所使用的默认配置，其中 L1Cache 和 L2Cache 均为私有 Cache，LLC 即 L3Cache 为多核心共享的末级 Cache。由于本文的评测重心在 L2Cache，因此仅仅为 L2Cache 配置了数据预取器，表 4.1 中也给出了数据预取器的默认配置。此外，本文为 Gem5 配置了 16 个 O3 核心，表 4.2 也给出了 CPU 相应的配置与校正情况。

表 4.2 评测使用的 CPU 配置及校正

配置参数说明	默认参数	校正后参数
核心数	16	—
核心类型	DerivO3CPU	—
Fetch Width	8	12
Fetch Queue Size	32	72
Decode Width	8	10
Issue Width	8	10
Writeback Width	8	10
Load Queue Size	32	72
Store Queue Size	32	44
Physical Int Reg Count	256	168
Physical Float Reg Count	256	160
Physical Vector Reg Count	256	168
Inst Queue Size	64	192
ROB Size	192	192
Branch Predictor	TournamentBP	Hashed Perceptron BP
RAS Size	16	32
Int ALU Count	6	4
Float ALU Count	4	2

## 4.1.2 评测程序

表 4.3 SPEC2006 评测程序的基本信息

测试程序名称	编程语言	说明	类型	16 进程
400.perlbench	ANSI C	Perl 语言	INT	√
401.bzip2	ANSI C	文件压缩与解压缩	INT	√
403.gcc	C	C 语言编译与优化	INT	√
429.mcf	ANSI C	公共交通的车辆调度	INT	×
445.gobmk	C	人工智能：围棋	INT	√
456.hmmmer	C	基因序列搜索	INT	√
458.sjeng	ANSI C	人工智能：国际象棋	INT	√
462.libquantum	C99	物理：量子计算	INT	√
464.h264ref	C	视频压缩与编码	INT	√
471.omnetpp	C++	离散时间仿真	INT	√
473.astar	C++	游戏人工智能：2D 寻路	INT	√
483.xalancbmk	C++	XML 到 HTML 的转换	INT	√
410.bwaves	Fortran 77	流体力学计算	FP	×
416.gamess	Fortran	量子化学	FP	×
433.mile	C	量子力学	FP	×
434.zeusmp	Fortran 77 & Real*8	物理：计算流体力学	FP	×
435.gromacs	Fortran & C	生物化学/分子力学	FP	√
436.cactusADM	Fortran 90 & ANSI C	物理：广义相对论	FP	×
437.leslie3d	Fortran 90	流体力学	FP	√
444.namd	C++	生物分子模拟计算	FP	√
447.dealII	C++	自适应有限元与误差分析	FP	√
450.soplex	ANSI C++	线性方程优化求解	FP	√
453.povray	ISO C++	计算机视觉：光线追踪	FP	√
454.calculix	Fortran 90 & C	结构力学	FP	×
459.GemsFDTD	Fortran 90	有限差分域求解麦克斯韦方程	FP	√
465.tonto	Fortran 95	量子晶体学	FP	×
470.lbm	ANSI C	流体动力学	FP	√
481.wrf	Fortran 90 & C	天气预测	FP	√
482.sphinx3	C	语音识别	FP	√

在评测程序的选用方面，本文从 SPEC2006 的基准评测程序的整点评测程序（INT2006）和浮点评测程序（FP2006）中，选择了 21 个在 16 个程序并行执行的情况下，内存使用量不会超过 8G 默认配置的测试集进行测试。表 4.3 给出了 SPEC2006 中的所有程序信息，其中 16 进程一栏表示并行执行 16 个进程时，测试程序是否会因内存占用超过 8G 而出错，标记“√”的表示不会出错，否则表示会出现错误。表 4.4 给出了本次测试程序所使用的编译器版本和编译信息。在工作集方面使用的是最大的 Reference 工作集，执行时选择热点代码执行 1000 万条指令，使用额外的 200 万条指令进行预热。由于本文所有的实验并未执行完整的程序，因此实验结果与整个程序的特征关系较小，与执行部分代码的特征关系更大，后续的所有分析都是建立在执行部分代码的特征上的。

表 4.4 评测程序的编译环境

编译器	编译器版本	编译类型	编译优化级别
g++	7.1.0	64bit	-O3
gcc	7.1.0	64bit	-O3
gfortran	7.1.0	64bit	-O3

由于评测多线程程序需要在全系统（FS）模式下运行 Gem5 的模拟过程。但是原生 Gem5 在使用 O3CPU 的多核环境下，进行全系统模拟存在大量的错误，因此本文并没有选择多线程程序进行评测。本文选择更为稳定可靠的多进程并行模拟，在多核场景的实验中，多个进程会被分配到不同的核心上执行。相比于多线程模拟，多进程并行模拟的进程之间没有共享数据，这就使得部分预取相关的场景不会出现，但并不影响本文实验分析的核心内容。

表 4.5 宿主服务器配置表

服务器配置信息	配置参数
Server	Dell PowerEdge R730
Architecture	X86-64
CPU cores	48
CPU Info	Intel® Xeon® CPU E5-2650 v4 @ 2.20GHz
L1 Cache	32KB（DCache）/32KB（ICache）
L2Cache	256KB
L3Cache	30MB
Mem	125GB
System	Ubuntu 16.04（4.15.0-96-generic x86-64）



### 4.1.3 评测硬件环境

本次测试使用的宿主机是 Dell PowerEdge R730 服务器，表 4.5 给出了对应的详细配置信息。

## 4.2 性能评估标准

除了使用传统的 IPC 作为性能评价指标以外，本文还通过预取的覆盖率和准确率来衡量预取机制的性能。同时本文设计的预取统计分析模块也可以给出大量有用的统计数据，这些数据对分析性能的变化也是十分有帮助的。

### 1) IPC

$$IPC = \frac{n_{insts}}{t_{cycles}} \quad (4-1)$$

IPC 表示的是每个时钟周期平均能运行的指令个数，它能最直接的反映 CPU 运行某一个程序的速度，因此该指标被广泛的用于衡量处理器的性能。

### 2) 预取覆盖率

$$coverage = \frac{miss_{nopref} - miss_{pref}}{miss_{nopref}} \quad (4-2)$$

预取覆盖率常常被用来衡量一个预取器生成的预取请求可以覆盖多少程序的访问数据，或者说消除了多少次访问缺失。在计算覆盖率的时候，使用的是所有 Cache 中访问缺失次数的总和。预取覆盖率可以在一定程度上体现预取的性能提升，但由于预取覆盖率只是一个粗略的估算，因此预取覆盖率的提升比例和性能的提升比例往往不成严格的线性相关关系。

### 3) 预取准确率

$$accuracy = \frac{pref_{useful}}{pref_{total}} \quad (4-3)$$

相比于覆盖率，预取准确率更容易理解，它表示了预取器生成的请求中 useful 预取的数目。预取准确率可以用来衡量预取对带宽的浪费程度和对 Cache 的污染程度，准确率越高，造成的 Cache 污染越小，浪费的带宽也越少。

## 4.3 多核预取干扰分析

$$\delta_{original-n} = \frac{CPI_{nopref-1} - CPI_{nopref-n}}{CPI_{nopref-1}} \quad (4-4)$$

$$\delta_{pref-n} = \frac{(CPI_{pref-1} - CPI_{pref-n}) - (CPI_{nopref-1} - CPI_{nopref-n})}{CPI_{nopref-1}} \quad (4-5)$$

为了有效区分预取在多核场景下产生的干扰，本文将干扰分为原始干扰（ $\delta_{original-n}$ ， $n$  个进程并行时的原始干扰）和预取干扰（ $\delta_{perf-n}$ ， $n$  个进程并行时的预取干扰）两种：前者表示在没有开启预取器的情况下，并行运行多个进程对性能产生的影响；后者表示在开启预取器的情况下，并行运行多个进程时预取对性能产生的影响。式（4-4）与式（4-5）分别给出了并行运行  $n$  个进程时，原始干扰和预取干扰的计算方法，其中 CPI 表示平均每执行一条指令所需要的周期数。这两个公式只提供一种简单评估的方法，并不能直接代表实际干扰产生的性能损耗大小。原始干扰主要来自于不同进程之间对共享资源，诸如 IO 带宽、LLC 等的争用。预取干扰来自于共享资源的争用，同时过多的预取会导致需求请求响应的延迟。除此之外，预取之间也会因争用共享资源产生干扰，这些干扰会导致预取的时效性变差。

图 1.3 展示了不包含本文设计的预取控制器情况下，并行运行多进程时，干扰最为严重的几个评测程序的干扰情况。可以看出，bzip2、sjeng、lbm 和 gobmk 的预取较为严重，最多可以导致 40% 以上的性能损失。相同情况下，原始干扰产生的性能损失往往比预取干扰的性能损失更多。

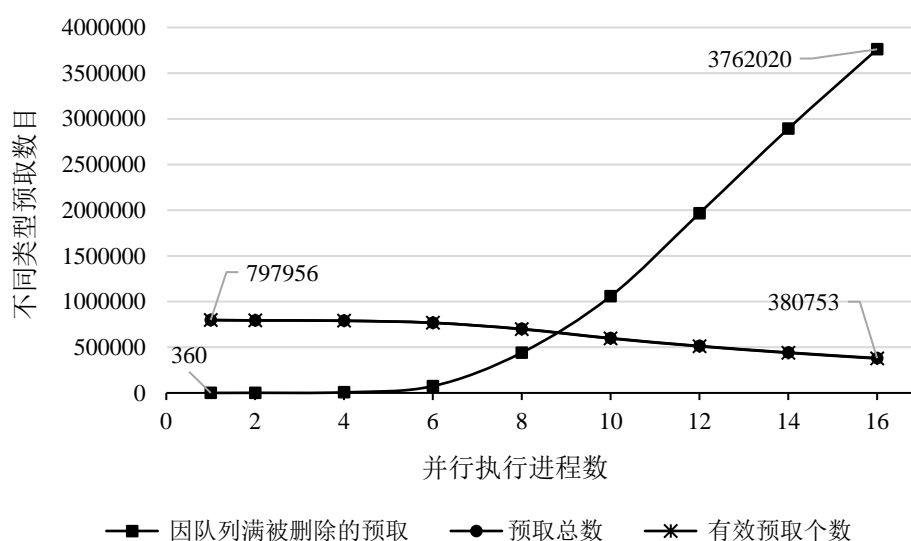


图 4.1 bzip2 中不同类型的预取个数

为了确定预取产生干扰的根本原因，本文对预取干扰严重的几个评测程序进行了分析，并对对多组统计变量进行了比较。以图 1.3 中预取干扰最严重的 bzip2 为例，图 4.1 中展示了 bzip2 测试中几种不同类型预取的个数随并行运行进程个数的变化情况。可以看出，在运行的这一段测试中，无论是 16 进程并行执行时的 98.93%，还是单进程执行时的 99.93%，预取的准确率都是很高的。在预取准确率下降很少的情况下，预取

总数和有效预取个数却有明显的下降。

由于预取缓冲队列的大小有限，如果队列已满，那么新生成的预取请求会将队列中最早的替换掉。由图 4.1 可以发现，因为预取缓冲队列已满而被删除的预取请求个数随进程数目增长速度十分迅速。在 16 个进程并行执行的情况下，因预取请求队列已满而被删除的预取个数可以达到单进程时的 10450 倍，预取总数的 10 倍。可见预取总数的下降很大程度上，是和多进程并行执行导致预取难以正常发送到 L2Cache 得到处理直接相关的。

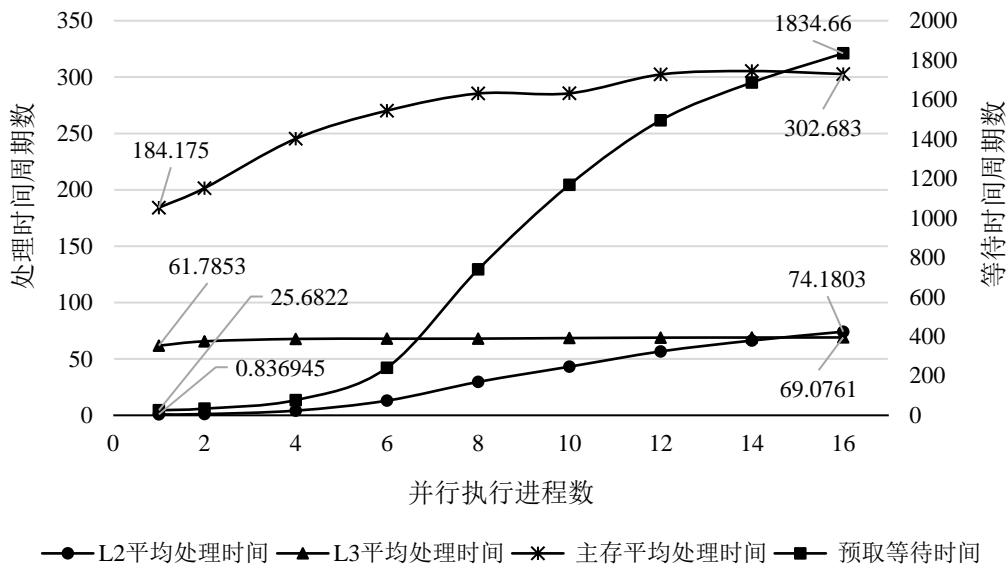


图 4.2 bzip2 中预取不同阶段的处理时间

为了进一步确定干扰的原因，本文也对 bzip2 中预取的处理时间进行了分析，图 4.2 给出了相应的结果。可以看到，预取请求在预取缓冲队列中的等待时间随进程数增长的变化十分明显，在 16 进程并行执行的时候，预取在缓冲队列中的平均等待时间已经达到了 1834 个周期，相比于单进程时的 25 个周期增加了约 73 倍。等待时间的增加以及因为队列已满而被删除预取个数的增加，都是预取难以发出得到处理的直接结果。而预取难以发出得到处理，最直接的原因便是 L2Cache 没有足够的 MSHR 表项分配给预取。如果 L2Cache 中 MSHR 的表项资源十分紧张，相应的处理时间也一定会增加。由图 4.2 也可以观察到，L2Cache 和 DRAM 的处理时间随着并行执行进程数目的增多都有明显地增长。但是 L3Cache 处理时间的变化却不大，因此预取干扰的增长和 L3Cache 关系并不大。

L2Cache 的响应速度和 L2 的 MSHR 表项数目直接相关，而 DRAM 的处理时间则和 DRAM 的通道数、行选延迟等参数密切相关。同时增大预取缓冲队列的大小也可能有效的减少预取干扰。图 4.3 分别给出了并行执行 16 个进程、4 个进程和单个进程时，增加 L2 中 MSHR 表项数目、增加 DRAM 的通道数目、增加预取缓冲队列大小对 bzip2

测试程序 IPC 的影响。可以看到增加 L2 中的 MSHR 表项或者增大预取缓冲队列大小，在多线程并行执行时都未能有效地提升 bzip2 的性能。相对的，增加 DRAM 的通道数目，即增加 DRAM 的带宽，可以有效地提升 bzip2 在多线程并行执行时的性能。在并行执行 16 进程时，将 DRAM 通道数从 1 增加到 4 可以获得 62.55% 的性能提升。

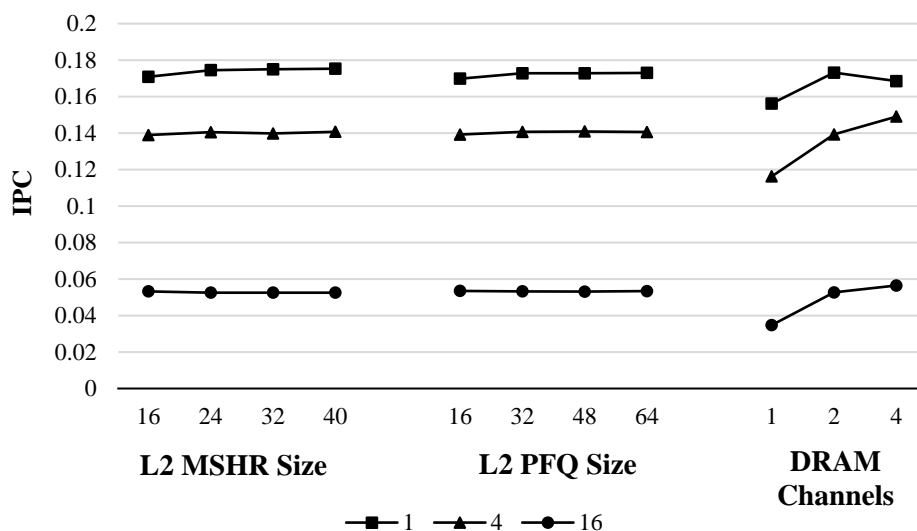


图 4.3 不同参数对 bzip2 性能的影响

如果 DRAM 带宽是预取干扰的核心原因，在理想情况下，4 通道并行执行 4 进程时的性能应该与单通道执行单进程的性能相当。但是图 4.3 给出的结果显示，4 通道并行执行 4 进程时的性能，只有单通道执行单进程时性能的 95.51%。这是因为通道数目的增加也会带来一定的通道调度开销，加上内存其他参数的限制和实际数据访问场景的影响，使得通道数增加 2 倍后，实际的带宽增加往往不足 2 倍。实际测试时统计到的，4 通道 4 进程时的 DRAM 预取平均处理时间是 221.65 个周期，单通道单进程时则是 210.55，是前者的 94.99%，和性能的变化比率十分接近。

综合上述分析，可以确定预取干扰的主要是由 DRAM 带宽不足导致的。DRAM 的带宽不足导致多线程并行执行时，难以快速的处理成倍增加的访问请求。访问请求处理的延迟，使得 L2Cache 中 MSHR 表项从分配到释放的时间间隔变长。L2Cache 中 MSHR 表项处理速度变慢，使得预取器难以为新的预取分配 MSHR。频繁的 L2Cache 访问失效触发了新的预取，也使生成的预取数目极速增长，实际得到处理的有效预取数目快速下降。这些因素最终导致了包含数据预取器的情况下，多线程并行执行时整体性能的严重下降。

#### 4.4 数据预取控制器的性能分析

本章节将分别从单进程运行场景，即单核场景和多进程并行场景，即多核场景分

别对数据与去控制器的性能进行评测与分析。

#### 4.4.1 单核场景性能评测

如果没有将特定配置作为实验变量，那么本文所有针对数据预取控制器的测试都是在默认配置下进行的。默认配置可以参照表 3.1、表 3.10、表 3.11 以及表 3.12 中的数据，其中预取过滤器模块并没有开启多核心场景下的共享，即每一个 L2Cache 的数据预取器都会配置一个单独的预取过滤器模块执行预取过滤操作。

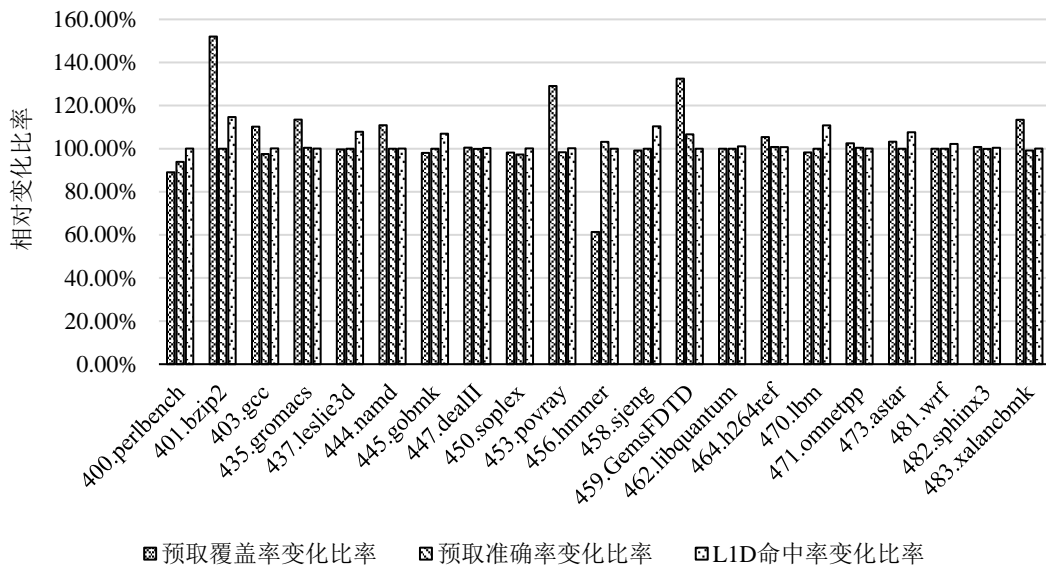


图 4.4 单核场景下预取属性的变化

虽然本文主要的工作是设计面向多核处理器的数据预取控制器，但由于本文对数据预取控制器的部分优化也适用于单核场景，因此在单核场景下也有着明显的性能提升。图 4.4 给出了单核（多核单进程）场景下使用本文设计的数据预取控制器时，预取覆盖率、预取准确率和 L1D 命中率的变化情况。从测试结果可以看到，在同时使用数据预取器和数据预取控制器的情况下，相比于仅使用数据预取器时，预取覆盖率平均<sup>①</sup>提升了 **4.17%**，L1D 命中率平均提升了 **2.94%**，但是预取准确率平均下降了 **0.15%**。

在预取准确率方面，数据预取器控制器的主要目的是动态控制预取请求的目标层级或过滤预取。预取等级的变更并不会导致一个无用预取变成有效预取，加上本文设计的数据预取器控制器支持预取无效化传递，所以如果被过滤的预取请求占比较小的话，预取准确率并不会因为数据预取控制器的加入而得到明显提升。相对的，由于支持预取目标等级的提升，部分准确预取在被提升到 L1D 时，相比于 L2 更容易被替换而变成无用预取，因此预取准确率会有一定的下降。但是数据预取器控制器是依据全局

<sup>①</sup> 如未做专门说明，所有相对变化比率数据的平均值均为几何平均值，其他平均值均为算术平均值

的预取事件进行动态训练，并依据训练结果对预取目标 Cache 层级进行控制的，因此预取准确率下降比率很小。

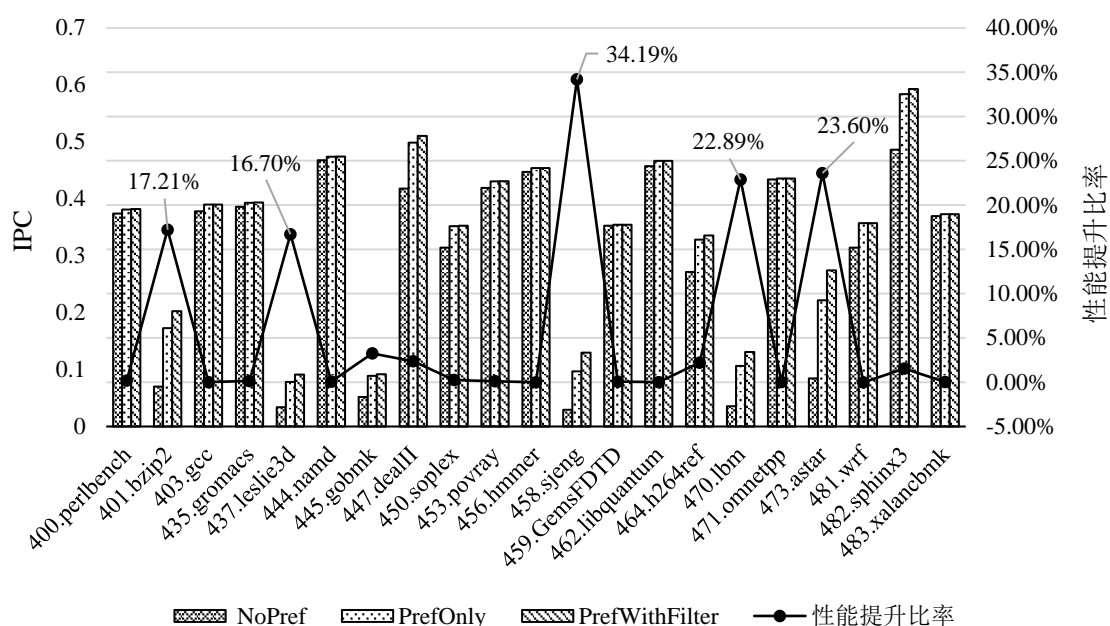


图 4.5 单核场景下的性能提升

图 4.5 给出了单核场景下，数据预取控制器对性能的提升情况。图中对比了没有预取（NoPref）、只有数据预取器（PrefOnly）和同时配置数据预取器及数据预取控制器（PrefWithFilter）三种情况下的 IPC。性能提升比率表示数据预取控制器所带来的，相对于只有数据预取器情况下的性能提升百分比。从测试结果可以看到，在同时使用数据预取器和数据预取控制器的情况下，相比于无预取可以获得平均 **43.84%** 的性能提升。相比于只有数据预取器的情况可以得到平均 **5.60%** 的性能提升。在图 4.5 中给出的所有评测程序中，除了 456.hmmmer 和 481.wrf 分别有 0.01% 和 0.02% 的性能下降外，其他所有的程序均有性能提升。401.bzip2、437.leslie3d、458.sjeng、470.lbm、473.astar 可以从本文设计的数据预取控制器中获得更多的性能提升，提升百分比在 **15% 到 35%** 之间。

除了基本的性能测试外，本文也为数据预取控制器添加了预取统计分析模块，可以对不同类型的预取进行统计。图 4.6 展示了评测程序在单核场景下，同时使用数据预取器和数据预取控制器时，不同类型的预取占比、发出预取总数以及访存指令占比的变化情况。从图中展示的结果可以发现，访存指令占比较高的访存密集型程序，发出的预取总数往往会更高。由于实验中使用的是基于固定步长的预取器，在数据访问模式比较规则的程序中往往会生成更多的预取。因此生成的总预取数目与访存指令占比、程序数据访问类型及数据预取器类型均有关系。在不同类型的预取占比方面，可以发现生成预取数目较少的程序中出现无用预取及有害预取的比例更大一些。在测试的 21 个程序中，平均约有 0.61% 的有害预取和 5.39% 的无用预取，有用预取占比较大。有害

预取的平均占比很小，因此有害预取的影响相比无用预取要小很多。如果作为基准的数据预取器自身的准确率就很高，那么有害预取和无用预取的影响也会进一步减小。

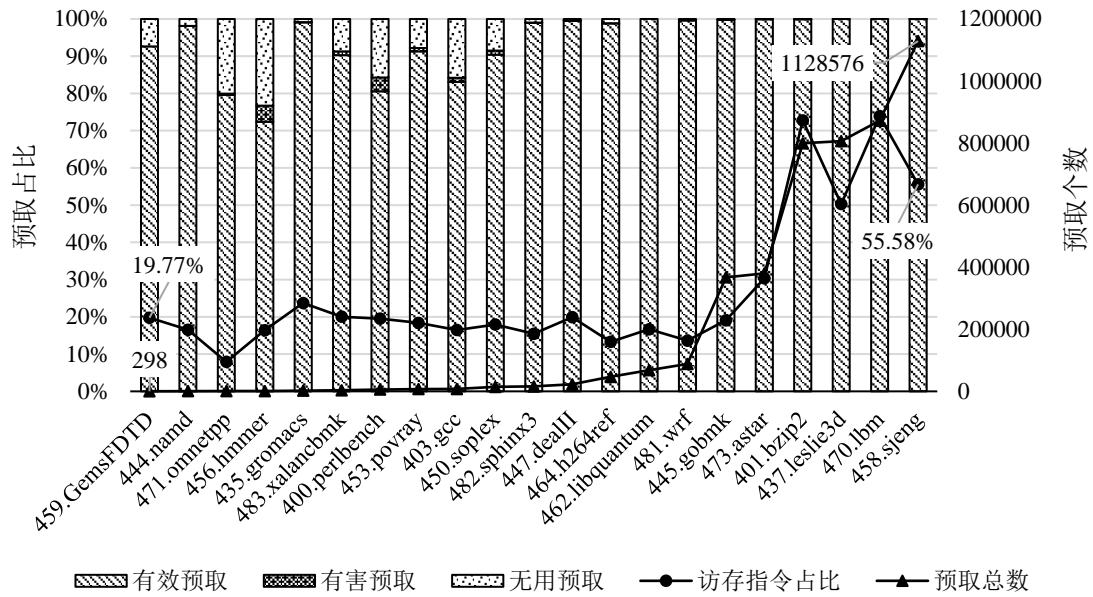


图 4.6 单核场景下不同类型预取占比

为了进一步确认不同程序从数据预取控制器获益的原因，本文对所有评测程序的统计变量进行分析，发现不同程序性能变化的原因差别较大。有的程序即使准确率与覆盖率都很高，也依旧没有获得明显的性能提升。表 4.6 给出了相应的分析结果。

表 4.6 数据预取控制器对性能影响的分析

评测程序名称	性能提升比率	访存指令占比	预取准确率	预取覆盖率	原因分析
400.perlbench	0.19%	19.53%	80.53%	1.88%	覆盖率低，基本达到性能提升上限；预取提级导致非有效预取个数增加
401.bzip2	17.21%	72.77%	99.92%	60.56%	准确率高且覆盖率高；大量有效预取被提升至 L1DCache，覆盖率得到明显提升，性能亦得到明显提升
403.gcc	0.01%	16.29%	83.08%	12.18%	计算密集型，覆盖率低，同时 L2 训练出的预取请求少，基本达到性能提升上限
435.gromacs	0.14%	23.71%	99.08%	25.26%	L1 数据复用率高，L2 训练出的预取请求少，基本达到性能提升上限
437.leslie3d	16.70%	50.30%	99.98%	70.11%	覆盖率高且准确率高；大量有效预取被提升至 L1DCache，性能得到明显提升

评测程序名称	性能 提升比率	访存指令 占比	预取 准确率	预取 覆盖率	原因分析
444.namd	0.06%	16.54%	98.12%	14.42%	计算密集型，预取覆盖率较低，L2 训练出的预取请求少，基本达到性能提升上限；
445.gobmk	3.25%	19.07%	99.82%	49.17%	访存操作主要是写操作，计算密集型，覆盖率不高，预取提级并不能获得较多的性能提升
447.dealII	2.36%	19.93%	99.55%	49.34%	计算密集型，预取覆盖率不高，L2 训练出的预取请求少，基本达到性能提升上限；
450.soplex	0.25%	17.94%	90.39%	24.53%	计算密集型，L2 训练出的预取请求少，基本达到性能提升上限；预取提级导致非有效预取个数增加
453.povray	0.11%	18.40%	91.30%	9.86%	计算密集型，覆盖率低，L2 训练出的预取请求少，基本达到性能提升上限
456.hmmmer	-0.01%	16.41%	72.38%	4.40%	计算密集型，覆盖率低，L2 训练出的预取请求少，预取提级导致非有效预取个数增加
458.sjeng	34.19%	55.58%	99.97%	85.50%	覆盖率高且准确率高；大量有效预取被提升至 L1DCache，性能得到明显提升
459.GemsFDTD	0.06%	19.77%	92.62%	2.45%	计算密集型，覆盖率低，同时 L2 训练出的预取请求少，基本达到性能提升上限
462.libquantum	0.01%	16.67%	100%	86.08%	访存操作中写操作占据一半，预取提级获得性能提升有限
464.h264ref	2.22%	13.29%	98.81%	66.36%	计算密集型，覆盖率高，但 L2 训练出的预取请求数目有限，性能提升受到限制
470.lbm	22.89%	73.90%	100%	80.28%	覆盖率高且准确率高；大量有效预取被提升至 L1DCache，性能得到明显提升
471.omnetpp	0.02%	7.91%	79.59%	4.91%	计算密集型，覆盖率低，同时 L2 训练出的预取请求少，基本达到性能提升上限
473.astar	23.60%	30.36%	99.92%	86.35%	覆盖率高且准确率高；大量有效预取被提升至 L1DCache，性能得到明显提升



评测程序名称	性能 提升比率	访存指令 占比	预取 准确率	预取 覆盖率	原因分析
481.wrf	-0.02%	13.62%	99.63%	75.67%	访存操作主要是写操作，预取提级并不能获得较多的性能提升，同时预取提级导致非有效预取个数增加
482.sphinx3	1.54%	15.46%	99.02%	59.73%	计算密集型，覆盖率高且准确率高，但 L2 训练出的预取请求数有限，性能提升受限
483.xalancbmk	0.03%	20.05%	90.29%	5.82%	覆盖率低，L2 训练出的预取少，基本达到性能提升上限

从分析结果可以发现，如果要通过数据预取控制器获得较大的性能提升，需要满足几个必要条件：访存密集型程序，数据预取器的覆盖率和准确率较高。数据预取控制器能够提升多少性能和作为基准的数据预取器密切相关。如果基准的数据预取器在一个访问密集型程序中只能生成很少的有效预取，那么即使加入数据预取控制器，性能提升幅度也会受到严重的限制。

如果访存指令以写操作为主，那么将数据预取到高层级 Cache 所产生的性能提升较小。这是因为当即将写入的数据不存在于 L1DCache 中时，Cache 会分配写缓冲区存放数据，后续对其中的数据读写操作大多会直接命中。如果写入的数据已经存在于 L1DCache 中，那么写操作命中缓存行并写入，后续的操作和前者差别较小。但是相比于 Cache，DRAM 并没有写缓冲区，因此在 DRAM 中处理的写操作会导致后续的读写操作产生较大的等待延迟。同时盲目的将写操作相关的预取提升至 L1DCache，可能会对 Cache 造成污染。因此在一般情况下，针对写操作预取的目标层级变更所获得的性能提升，没有读操作预取的高。

经过分析可以发现多数性能提升较高的程序主要得益于预取目标层级的提升，而非有效预取的过滤并非性能提升的主要因素。图 4.7 给出了禁用预取有害性统计和训练以及禁用预取目标层级提升后的性能变化情况。可以发现 PPF 原生的过滤功能可以获得平均 0.02% 的性能提升，而添加预取有害统计功能可以获得平均 0.03% 性能提升，预取目标层级的扩展可以获得平均的 5.57% 性能提升。

从单个评测程序来看，有害性统计对 447.dealII 的支持更好，可以带来 0.34% 的性能提升，在 403.gcc 这种预取占比较小的计算密集型程序中则产生了 0.16% 的性能下降。在 458.sjeng 中，使用完整预取器比使仅使用预取提级功能的性能提升了 0.25%，这是因为预取层级的提升可能更容易在 L1DCache 中引入有害或者无用预取，有害统计支持可以避免这种现象产生性能损失。总而言之，有害性统计能否带来足够的性能提升与基准预取器的关系较大，使用覆盖率更大的数据预取器可以更大程度的发挥数据预取控制器的效用。

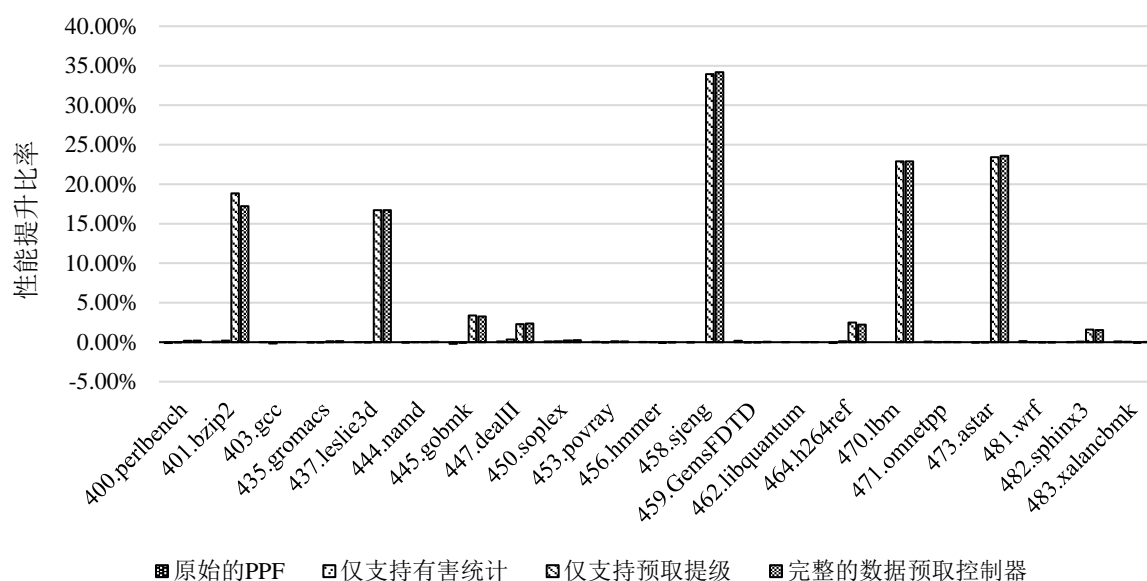


图 4.7 不同配置下数据预取控制器的性能提升

从图 4.7 整体来看，数据预取控制器带来的性能增长主要来自于预取目标 Cache 层级提升的支持。前面的章节提到过，预取目标层级变更有着不同的实现方法。相比于将预取直接发送到目标层级 Cache 的预取缓存队列中，本文的实现方法可以避免不同层级 Cache 之间因预取请求的传递产生的严重 IO 开销。从另一个方面讲，使用预取控制器将预取数据存放到高层级 Cache 中和直接在高层级 Cache 中添加数据预取器也有着本质的区别。由于数据预取器发出得预取请求需要占用 MSHR 表项，而 L1DCache 中的 MSHR 表项数目要比 L2Cache 中少很多。加上 L2Cache 接收到的请求均为缓存行大小对齐的访问，MSHR 中 Target 位置的利用率也会比 L1DCache 更高。所以直接在 L1DCache 中添加数据预取器，会对 MSHR 带来更多的压力。

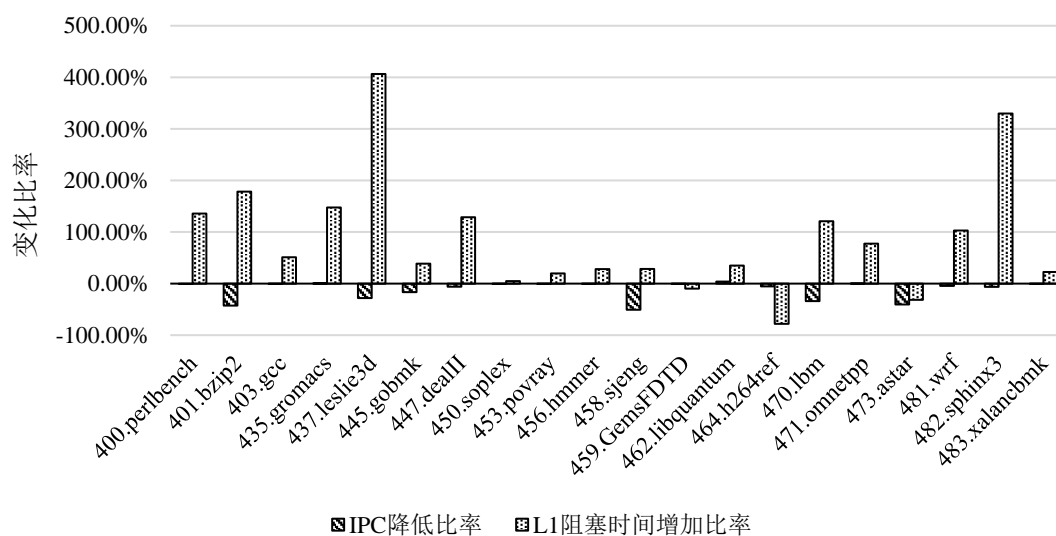


图 4.8 L1DCache 使用数据预取器的性能变化

图 4.8 便给出了不同程序在 L1DCache 中添加数据预取器相对于本文设计的性能变化,同时也给出了 L1DCache 因为 MSHR 已满或单个 MSHR 表项存放的 Target 已满,而不能接收新的请求,导致 Cache 阻塞的周期数目变化。其中 IPC 平均下降了 12.95%, L1 因为 MSHR 阻塞的时间增长了 78.10%。可以看出本文的实现方法,可以有效避免对高层级 Cache 中 MSHR 的占用,从而避免 Cache 的阻塞,在性能提升上具有更好的效果。

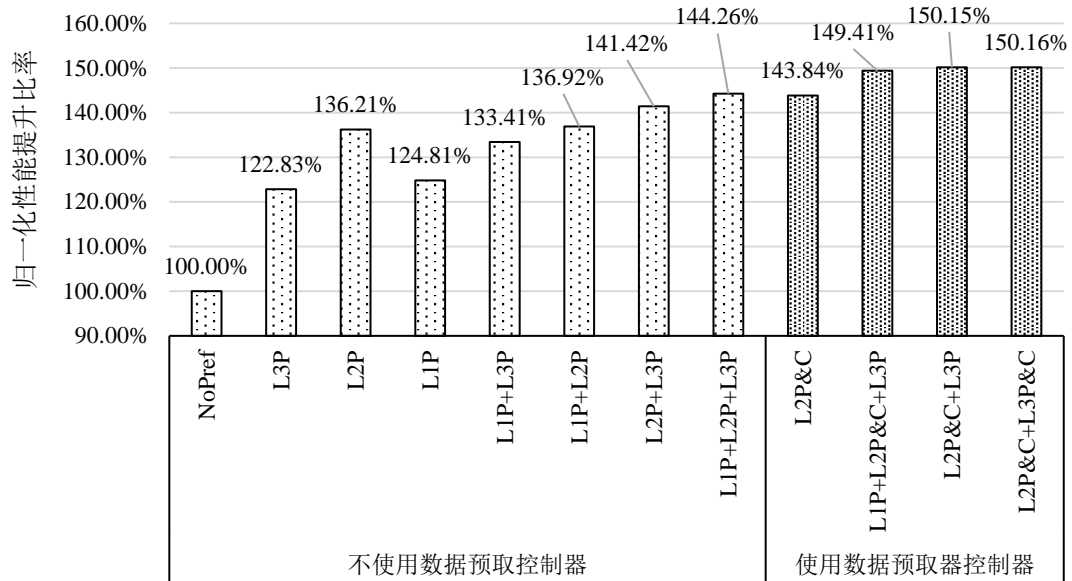


图 4.9 不同 Cache 配置下的性能变化情况

在实际的处理器中常常会在不同层级使用预取器来获取更高的性能提升比率,因此本文也对不同预取器配置下,以及不同配置使用预取控制器的情况下的性能进行了测试和对比。图 4.9 展示了不同配置下的性能归一化提升比率,其中后缀“P”表示当前层级 Cache 中设置使用默认配置的预取器,后缀“P&C”表示当前层级不仅设置了预取器,还配置了相应的预取器过滤器模块对预取器的预取请求进行过滤。由于使用预取控制器场景下的配置类型过多,因此这里仅仅展示了其中性能最好的三个配置以及本文的默认配置。

由图 4.9 展示的结果可以发现,使用预取控制器相比于同样配置下不使用预取控制器,都会有明显的性能提升。比如“L2P&C+L3P&C”的配置性能相比于“L2P+L3P”平均提升了 6.18%,即使相对于无预取控制器时性能最好的“L1DP+L2P+L3P”,也有平均 4.09%的性能提升。“L2P&C+L3P”和“L2P&C+L3P&C”两种配置的性能相差很小,这是因为本文默认配置的预取过滤阈值很小, L3 预取器发出预取请求被过滤掉的比率很低,对性能影响较小。从另一方面看,使用“L2P&C+L3P”的配置不仅可以获得较好的性能提升,还可以避免为 L3 预取器配置预取过滤器,从而进一步节省了存储开销。相比于本文的默认结构配置“L2P&C”,是更适合实际处理器使用的配置方法。

#### 4.4.2 多核场景性能评测

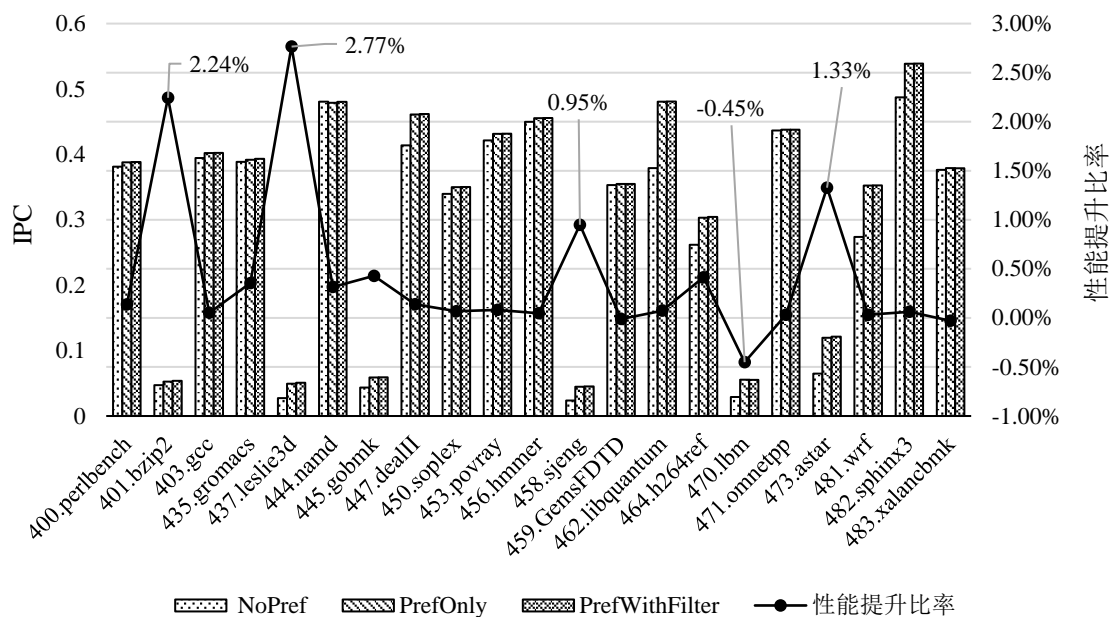


图 4.10 16 核 16 进程场景下的性能提升

相比于单核场景，16 进程的多核场景下数据预取控制器所获得性能提升下降了很多，平均只有约 0.43% 的性能提升。如图 4.10 所示，单核场景下性能提升超过 15% 的评测程序，在 16 进程的多核场景下的平均性能提升不足 3%。其中 470.lbm 甚至出现了约 0.45% 的性能下降，主要是因为提级预取导致 L2 训练预取器的事件变少，预取器生成预取数量变少导致的。实际上其他测试集或多或少也会出现改问题，但是 lbm 预取器生成预取的减少量要大得多。生成预取减少所产生的性能损失，掩盖了预取目标层级提升所带来的性能提升。多核场景下预取干扰的增加，导致实际发出的预取数目锐减，也是性能提升下降的原因之一。

由于 LLC 中数据复用率低，且本文设计的数据预取控制器支持预取无效化的向下传递，所以在实际的测试中，并没有发现跨核心有害的预取。预取无效化向下传递是指高层级 Cache 中的预取数据被需求请求命中之后，低层级 Cache 中相同的预取数据，在特定的延迟后也会被清除预取属性以避免冗余的训练事件。这样的无效化传递操作是符合实际场景的，因为即使没有预取，需求请求也会将数据存放到低层级的 Cache 中，因此无效化之后预取的相关事件和预取本身并没有关系。这也使得使用准确率较高的预取器时，几乎所有预取在 LLC 中替换数据被命中之前，便都被无效化，没有统计到跨核心有害计数也是正常的情况。相比之下，单核场景下可以统计到有害预取，但这些有害预取多数出现在像 L1DCache 这样容量较小的高层级 Cache 中。

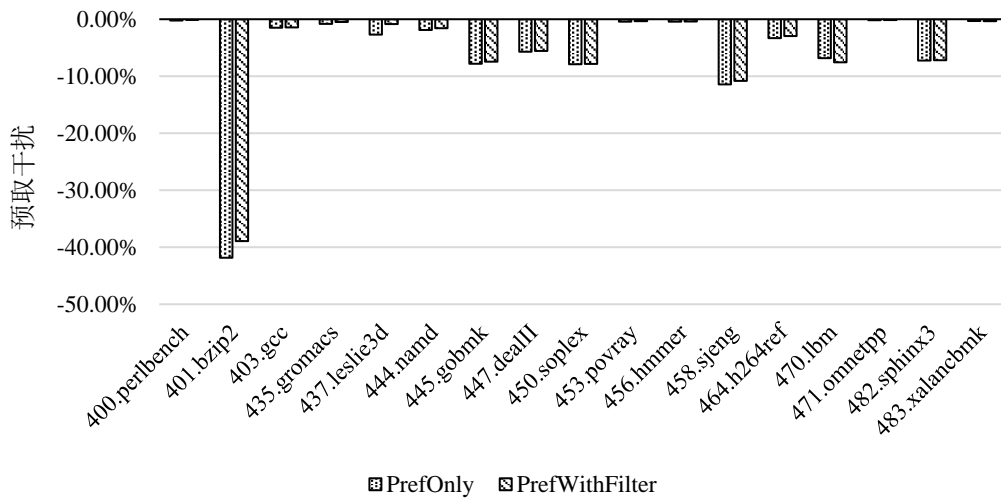


图 4.11 16 核 16 进程场景下预取干扰的变化

图 4.11 展示了 16 进程的多核场景下使用数据预取控制器与不使用时预取干扰的变化情况。使用数据预取控制器之后多数程序的预取干扰均有下降，平均降低到不使用数据预取控制器时的 81.53%。虽然使用数据预取控制器降低了预取干扰，但是实际的预取干扰依旧很大。本文设计的数据预取控制器对预取干扰的处理效果不佳主要有两个方面的原因：一个是预取训练的干扰，另一个则是特征对过滤结果贡献度不可变。

预取过滤器模块是基于感知器学习的方法，通过对预取不同的特征的权重进行训练，并基于权重均值实现预取过滤的。然而在预取器生成的一个预取访问流中的不同预取，会共享表 3.7 中展示的多个特征，即他们在这些特征上的数值是相同的。那么无论在训练还是过滤时，使用的特征权重表表项也是相同的。当一个预取访问流中的部分预取并非有效预取时，所产生的训练结果也会影响到同一个访问流中的其他预取，产生一定的训练污染。此外，虽然特征中包含了预取深度信息，但过滤方法决定了每一个特征值只能在最后的平均值贡献固定的一部分。对于一个满权重的预取，计算得到的权重均值等于 32，即使预取深度相关特征的权重变为 0，权重均值也只能降低至 28，因此并不能对深度较大的预取进行单独的过滤。所以使用感知器学习的方法进行预取过滤，更多情况下可以对不同的预取流进行过滤，但针对同一个预取流中不同深度预取请求的区分度要小很多，最终就会使得处理预取干扰的效果欠佳。

#### 4.5 配置对数据预取控制器性能的影响

为了减少数据预取控制器的存储开销，本文在设计时对所有组相联结构表格的 Tag 位数进行了缩减，使整体的存储开销节约了 38.92%。同时如图 4.12 所示的一样，将 Tag 位数减少至 6-bit 时，整体性能平均变化了 0.12%。图中有不少评测程序在压缩 Tag 之后性能有一定的提升，是因为组相联结构表格中 Tag 位数的减少，会导致不准确的命

中和替换，所带来的训练效果可能是正面的也可能是负面的。总的来说，缩减 Tag 位数可以在减少数据预取控制器的存储开销的同时，保证性能没有明显的变化。

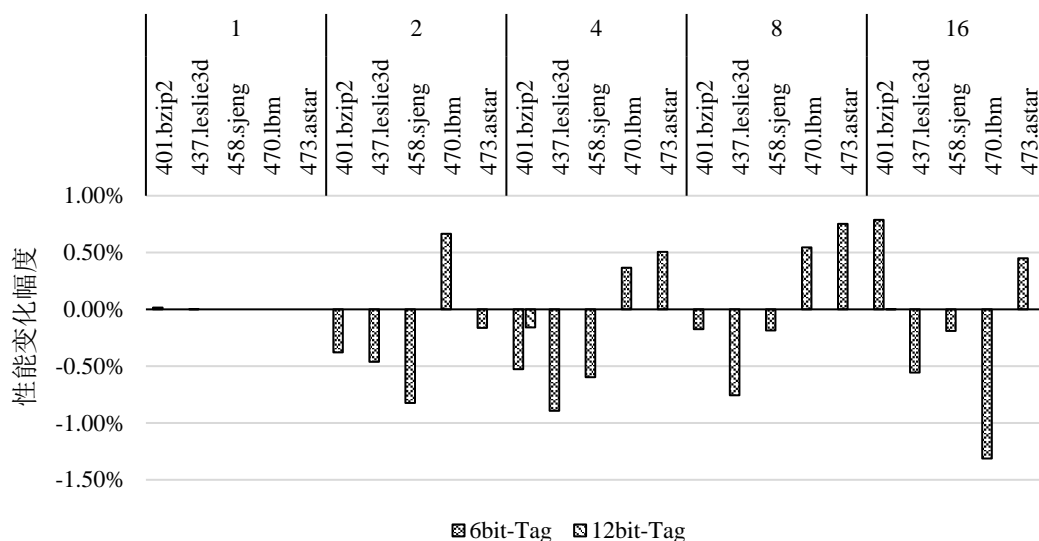


图 4.12 使用不同位数 Tag 对性能的影响

另一方面，预取过滤器模块中的有效预取记录表和过滤预取记录表大小也会对预取的训练效果产生影响。图 4.13 展示了单核场景下，不同大小的有效预取记录表和过滤预取记录表配置下的性能变化，所有的性能变化幅度都是基于默认配置的 1024 表项大小计算得到的。可以发现，减少表格的表项数目并不一定会产生性能的损失，这和使用不完整 Tag 时的性能变化原因是相同的。表格尺寸的变化平均只有 0.06% 的性能变化，即使在使用 128 个表项的表格时，也仅有平均 0.10% 的性能变化。因此针对高层级的 Cache，通过缩减表格大小节约存储开销，并不会对性能表现产生严重的影响。

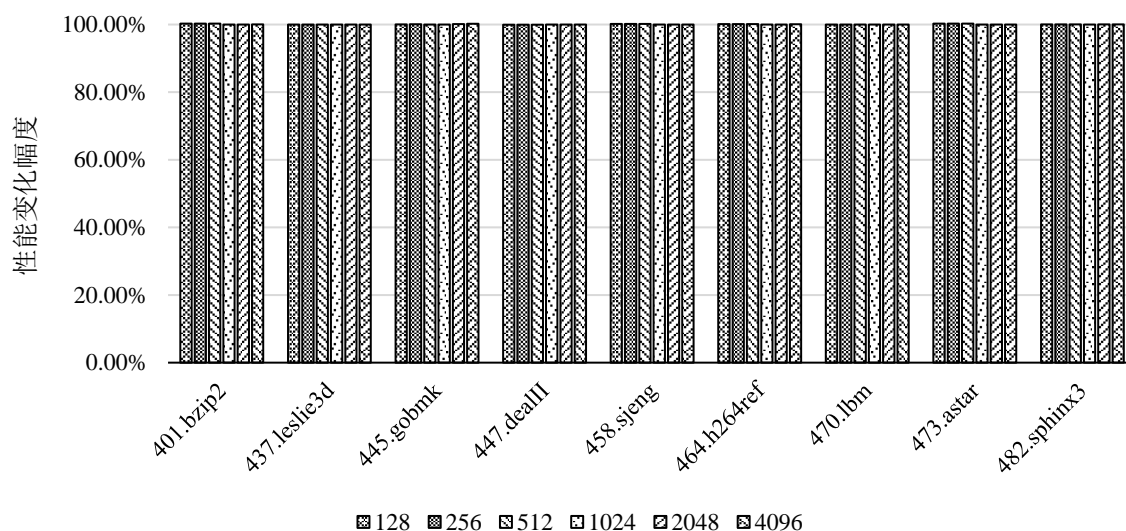


图 4.13 表格大小对性能的影响

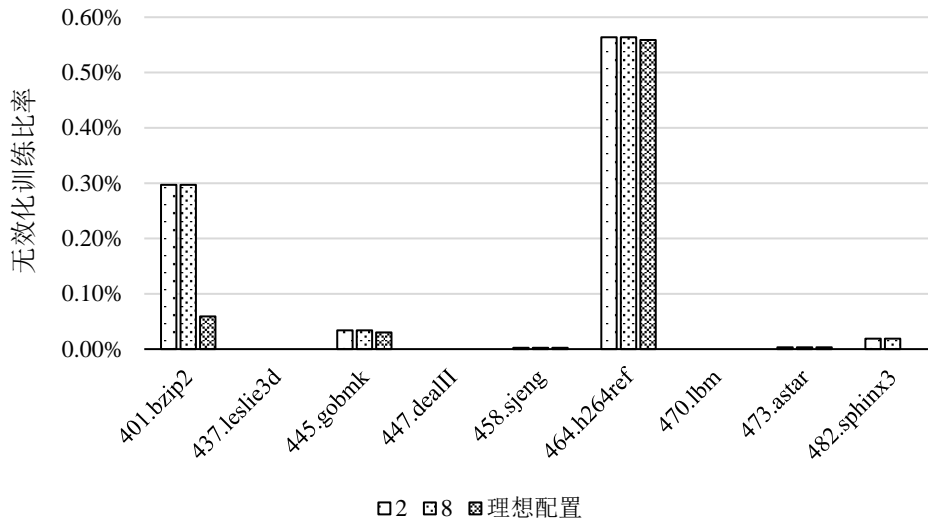


图 4.14 训练事件队列配置对训练的影响

由于本文将数据预取过滤器的训练场景扩展到了多核下，因此还设计了训练事件分发器进行训练事件的组织与分发。由于预取训练存在一定的延迟，以及同一个周期可能存在多个，具有相同目标的训练事件，该结构设计了训练事件队列来缓存无法及时得到处理的训练事件。然而训练事件队列的添加，依旧不能保证所有的训练事件不会被无效化。图 4.14 便给出了单核场景下，不同队列设计情况下无效化训练事件的占比变化情况。由图可见，配置变化对比率的影响十分微小。除了 401.bzip2 以外的评测程序中，相比于理想情况，使用大小为 2 的缓冲队列，无效化训练事件平均仅增加了 0.02%。其中 401.bzip2 无效化训练比率增加最多，也只有 0.24%。在性能方面，图 4.14 对应的评测程序在使用大小为 2 的缓冲队列时，相比于理想情况，平均只有 0.15% 的性能影响。因此本文默认配置下的训练事件分发器，在使用很小的存储开销时，并不会对性能产生严重的影响。

经过对不同配置测试可以发现通过进一步调整配置，可以在几乎不影响数据预取器控制器性能的情况下，进一步优化存储开销。如果我们将所有的表格大小设置为 128，平均性能变化不会超过 0.50%。同时有害性模块开销可以从 6 286 Bytes，减少到 606 Bytes，过滤器模块开销从 36 794 Bytes 减少至 16 970 Bytes。本文新增结构存储开销进一步降低至 **0.96KB**，仅占 L2Cache 大小的 0.37%。

## 4.6 本章小结

本章对数据预取控制器的性能进行了完整的分析与测试，同时借助预取统计分析模块确定了多核场景下预取产生干扰的主要原因。最后测试了数据预取控制器配置对性能的影响。从测试结果可以看出，本文实现的数据预取控制器在单核场景下，性能平

均提升了 5.60%，其中访存密集型程序性能平均提升了 22.76%；16 核场景下访存密集型程序的性能平均提升了 0.43%。



## 第五章 总结与展望

Cache 访问缺失会使处理器因等待数据而停顿流水线, 这个问题是限制处理器性能的主要因素之一。特别是在访存密集型的应用中, Cache 性能优化对处理器性能的提升起着至关重要的作用。数据预取技术通过提前将要访问的数据加载到 Cache 中, 来达到隐藏存储访问延时的效果。然而目前主流商用处理器中使用的多为单核场景下设计的预取器, 它们在多核场景下可能存在着干扰。同时单个预取器因为很难同时提升覆盖率和准确率, 在性能提升上遇到了瓶颈。本文在基于感知器设计的预取过滤器的基础上, 设计并实现了多核心场景下的数据预取控制器, 让数据预取的性能得到了进一步的提升。

### 5.1 本文主要工作

本文的主要工作包括下面三个方面:

- 1) 对数据预取器的发展情况进行了调研与分析。通过调研国内外学者在数据预取器设计思路上的发展, 对经典预取器优缺点进行了对比和分析。不同的数据预取器设计中包含不同的预取负面效应控制方法, 本文将该内容作为研究的核心进行了归类, 调研和分析了当前比较常见的预取控制设计思路。在实验环境方面, 整理了 ARM、Intel 和 AMD 主流商用处理器或微架构中的预取器设计, 发现主流商用处理器中的预取器多是基于单核场景设计。综合预取负面效应控制的归纳与对比, 选择了基于有效性统计对预取进行细粒度过滤的方法作为本文的主要实现方案, 并尝试将该设计适配到多核场景中。
- 2) 设计并实现了面向多核处理器的数据预取控制器。对 Gem5 模拟器中的 Cache 设计以及数据预取的处理流程进行了分析。设计并实现了准确的预取统计分析模块, 来对预取行为、有效性、有害性等特征进行统计和分析。为了获取更贴近实际的统计数据, 预取统计分析模块在无效化、预取合并的向下传递操作中保证了时序准确的实现。设计了可以统计预取有害性的相关模块, 可辅助数据预取控制器结构进行预取的惩罚性训练。设计与实现了预取过滤器模块, 并丰富了数据预取控制器的训练场景, 提升了训练效果, 在较低的存储开销下实现了多核场景下的扩展与优化。改进的数据预取控制器支持对预取的分类动态训练, 并基于设置的阈值对预取请求进行预取层级下降或者提升操作。
- 3) 数据预取控制器的性能评测与分析。本文基于预取统计分析模块的数据对预取器多核心干扰的情况进行了研究与分析, 确定了多核心预取干扰的主要原因。

使用 Gem5 的结构级模拟器及 SPEC2006 的评测程序对数据预取控制器进行了性能评测。结果显示单核场景下，性能平均提升了 5.60%，其中访存密集型程序性能平均提升了 22.76%；16 核 16 进程场景下访存密集型程序的性能平均提升了 0.43%。本文通过优化使整体存储开销降低了 38.92%，新增结构存储开销仅占 0.96KB，同时存储开销优化所带来的性能变化不足 0.2%。

## 5.2 未来工作展望

本文虽然通过将预取过滤器扩展到多核场景下，进行了一系列优化，并得到了性能提升。但是经过分析，发现多核场景下的预取干扰根本原因是受到了 DRAM 带宽限制，而单独通过预取有害性训练获得提升也十分微小。在传统的预取控制器设计中，会对预取器的激进度进行动态调整。但由于预取干扰的存在，预取过滤器基于激进度对应的特征值对预取进行过滤的方法，并不能有效的提升多核场景下的性能。因此在未来可以结合预取器激进度动态调度技术，进一步降低多核场景下的预取干扰，提升多核场景下的性能。

## 参考文献

- [1] Mark D. Hill. Amdahl's Law in the multicore era[C]. International Conference on High-performance Computer Architecture. 2008.
- [2] Erlin Yao, Yungang Bao, Guangming Tan, et al. Extending Amdahl's Law in the Multicore Era[J]. Performance Evaluation Review, 2009, 37(2):24-26.
- [3] Survey of Floating-Point Formats[OL]. <https://www.mrob.com/pub/math/floatformats.html>. 2020.
- [4] Patterson, David A, Hennessy, John L, Goldberg, David. Computer architecture: a quantitative approach[M]. Computer Architecture: A Quantitative Approach. Morgan Kaufmann Publishers Inc. 2008.
- [5] Shen, John Paul, Lipasti, Mikko H. Modern processor design : fundamentals of superscalar processors[M]. McGraw-Hill Higher Education, 2005.
- [6] Mittal S. A survey of recent prefetching techniques for processor caches[J]. ACM Computing Surveys (CSUR), 2016, 49(2): 35.
- [7] Mittal S.. A survey of architectural techniques for improving cache power efficiency[J]. Sustainable Computing: Informatics and Systems. 4, 1 (2014), 33–43.
- [8] Michael Ferdman, Cansu Kaynak, and Babak Falsafi. Proactive instruction fetch[J]. In International Symposium on Microarchitecture. 2011, 152–162.
- [9] Murali Annavaram, Jignesh M. Patel, and Edward S. Davidson. Data prefetching by dependence graph precomputation[J]. In International Symposium on Computer Architecture. 2001b, 52–61.
- [10] Srinath S., Mutlu O., Kim H., et al. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers[C]. 2007 IEEE 13th International Symposium on High Performance Computer Architecture. IEEE, 2007, 63–74.
- [11] Smith J. E., Hsu W. C.. Prefetching in supercomputer instruction caches[C]. Supercomputing '92. Proceedings. IEEE, 1992.
- [12] Intel 3B, Intel® 64 and IA-32 Architectures Software Developer's Manual Vol. 3B[R]. Order Number: 253669-059US. June, 2016.
- [13] Jouppi N.. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers[J]. Proc.annual Intl Sym.on Comp.archi, 1990.
- [14] Palacharla S., Kessler R. E.. Evaluating stream buffers as a secondary cache replacement[C]. International Symposium on Computer Architecture. IEEE, 1994.
- [15] Farkas K. I., Chow P., Jouppi N. P., et al. Memory-system design considerations for dynamically-scheduled processors[J]. Acm Sigarch Computer Architecture News, 1997, 25(2):133-143.
- [16] Luk, ChiKeung, Mowry, Todd C. Compiler-based prefetching for recursive data structures[J]. architectural support for programming languages & operating systems, 1996, 31(9):222-233.
- [17] Roth A., Sohi G. S.. Effective jump-pointer prefetching for linked data structures[C]. Computer Architecture, 1999. Proceedings of the 26th International Symposium on. ACM, 1999.

- [18] Z. Zhang and J. Torrellas. Speeding up irregular applications in shared-memory multiprocessors: Memory binding and group prefetching[J]. In 22nd Annual International Symposium on Computer Architecture, June 1995.
- [19] Bakhshalipour M, Shakerinava M, Lotfi-Kamran P, et al. Bingo spatial data prefetcher[C]. 2019 IEEE International Symposium on High Performance Computer Architecture (HPCA). IEEE, 2019: 399-411.
- [20] Shevgoor M., Koladiya S., Wilkerson C., et al. Efficiently Prefetching Complex Address Patterns[C]. IEEE/ACM International Symposium on Microarchitecture. ACM, 2015, 141–152.
- [21] Joseph D., Grunwald D.. Prefetching using Markov predictors[J]. Acm Sigarch Computer Architecture News, 1997, 25(2):252-263.
- [22] Michaud P. Best-offset hardware prefetching[C]. 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA). IEEE, 2016: 469-480.
- [23] J. Kim, S. H. Pugsley, P. V. Gratz, A. L. N. Reddy, C. Wilkerson, and Z. Chishti. Path confidence based lookahead prefetching[J]. 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2016, 1–12.
- [24] Bhatia E, Chacon G, Pugsley S, et al. Perceptron-based prefetch filtering[C]. Proceedings of the 46th International Symposium on Computer Architecture. ACM, 2019: 1-13.
- [25] SPEC CPU® 2017[OL], <http://www.spec.org/cpu2017/>, 2019.
- [26] Intel 64 and IA-32 architectures optimization reference manual[R]. Cooperation I, 2019.
- [27] Software Optimization Guide for AMD Family 15h Processors[R]. Devices A M, 2014.
- [28] Cortex-A65 Core: Technical Reference Manual[R]. Cortex ARM, Revision: r1p1, Feb, 2019.
- [29] SPEC CPU® 2006[OL], <https://www.spec.org/cpu2006/>, 2018.
- [30] Mahmut Kandemir, Yuanrui Zhang, and Ozcan Ozturk. Adaptive prefetching for shared cache based chip multiprocessors[C]. In Conference on Design, Automation and Test in Europe. 2009, 773–778.
- [31] Jiyang Yu, Peng Liu. A Thread-Aware Adaptive Data Prefetcher[C]. IEEE International Conference on Computer Design. IEEE, 2014.
- [32] Jorge Albericio, Ruben Gran Tejero, Pablo Ibáñez, et al. ABS: A Low-Cost Adaptive Controller for Prefetching in a Banked Shared Last-Level Cache[J]. Acm Transactions on Architecture & Code Optimization, 2012, 8(4):19.
- [33] Binkert N., Beckmann B., Black G., et al. The gem5 Simulator[J]. Computer architecture news, 2011, 39(2) : p.1-7.
- [34] Yedlapalli P., Kotra J., Kultursay E., et al. Meeting midway: Improving CMP performance with memory-side prefetching[C]. International Conference on Parallel Architectures & Compilation Techniques. IEEE, 2013.
- [35] Hur I., Lin C.. Memory Prefetching Using Adaptive Stream Detection[C]. 2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06). IEEE, 2006.
- [36] Stephen Somogyi, Thomas F. Wenisch, Anastasia Ailamaki, and Babak Falsafi. Spatio-temporal memory streaming[J]. In International Symposium on Computer Architecture (ISCA). 2009, 69–80.
- [37] David Kadjo, Jinchun Kim, Prabal Sharma, et al. B-Fetch: Branch Prediction Directed Prefetching for Chip-Multiprocessors[C]. IEEE/ACM International Symposium on Microarchitecture. IEEE, 2015.

- 
- [38] Victor Jiménez, Gioiosa R., Cazorla F. J., et al. Making data prefetch smarter: adaptive prefetching on POWER7[C]. International Conference on Parallel Architectures & Compilation Techniques. IEEE, 2012.
- [39] Sanyam Mehta, Zhenman Fang, Antonia Zhai, and Pen-Chung Yew. Multi-stage coordinated prefetching for present-day processors[C]. In International Conference on Supercomputing. 2014, 73–82.
- [40] 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 (HPCA). IEEE, 2014.
- [41] Carole-Jean Wu, Aamer Jaleel, Margaret Martonosi, Simon C. Steely Jr, and Joel Emer. PACMan: Prefetch-aware cache management for high performance caching[J]. In International Symposium on Microarchitecture. 2011, 442–453.
- [42] Jain A., Lin C.. Rethinking Belady's Algorithm to Accommodate Prefetching[C]. 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA). ACM, 2018.
- [43] Ebrahimi E., Mutlu O., Lee C. J., et al. Coordinated Control of Multiple Prefetchers in Multi-Core Systems[C]. 42st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-42 2009), December 12-16, 2009, New York, New York, USA. ACM, 2009.
- [44] Wu C. J., Martonosi M.. Characterization and dynamic mitigation of intra-application cache interference[C]. IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2011, 10-12 April, 2011, Austin, TX, USA. IEEE, 2011.
- [45] Alameldeen A. R., Wood D. A.. Interactions between compression and prefetching in chip multiprocessors[C]. 2007 IEEE 13th International Symposium on High Performance Computer Architecture. IEEE, 2007.
- [46] Dang X., Wang X., Tong D., et al. An adaptive filtering mechanism for energy efficient data prefetching[C]. Design Automation Conference. IEEE, 2013.
- [47] Guo Y., Narayanan P., Bennaser M. A., et al. Energy-efficient hardware data prefetching[J]. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 2009, 19(2): 250-263.
- [48] Kim, Taesu, Zhao, Dali, Veidenbaum, Alexander V. Multiple stream tracker: a new hardware stride prefetcher[C]. Acm Conference on Computing Frontiers. ACM, 2014.
- [49] The 2nd Data Prefetching Championship (DPC2)[OL], <http://comparch-conf.gatech.edu/dpc2/>, 2015.
- [50] Software and Infrastructure | DPC3[OL], <https://dpc3.compas.cs.stonybrook.edu/>, 2019.
- [51] ChampSim[OL], <https://github.com/ChampSim/>, 2020.
- [52] Jimenez D. A., Lin C.. Dynamic branch prediction with perceptrons[C]. International Symposium on High-performance Computer Architecture. 2001.



## 致谢

能够顺利的完成论文，首先要感谢我的家人在疫情期间的支持、鼓励和关怀。每次遇到困难，家人都会对我进行开导，让我有了继续工作的动力。如果没有他们，我可能很难坚持完成所有相关的实验以及论文的撰写。

感谢我的导师刘先华副教授，刘老师学识渊博、为人谦和、循循善诱，在我的学业上和生活中给予了我无微不至的指导和关怀，并在最后和易江芳老师、佟冬老师一同认真细致地指导我修改和完成了毕业设计论文。刘老师看待问题全面而且深刻，每每向刘老师请教问题都会让我茅塞顿开，三年来受益匪浅。

感谢程旭教授、易江芳老师、陆俊林老师、佟冬老师、刘锋老师在这三年硕士研究生生涯中的指点和帮助。程老师学术造诣高深，国家情怀浓重，深深影响了我和实验室的每一位同学。易老师、陆老师、佟老师和刘锋老师学术严谨、技术精湛，同时对同学们也十分关心，他们的一丝不苟的学术精神会一直激励着我。

感谢崔宏伟同学在处理毕业设计期间，不厌其烦的和我一起讨论学术问题和我对毕业设计提出的建议。感谢周昱晨、叶嘉成、戚妙、周叔欣等同实验室同学的关怀和帮助。感谢我的舍友们陪伴我在北大度过了三年开心快乐的生活。

能够在新型冠状病毒期间顺利完成论文，也要感谢那些奋战在一线的医护人员们。感谢你们，是你们让大家坚定了战胜病毒的信心，并一直健康快乐的生活着。相信中国一定可以打赢这一场对病毒的战役，中国加油！





# 北京大学学位论文原创性声明和使用授权说明

## 原创性声明

本人郑重声明：所呈交的学位论文，是本人在导师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不含任何其他个人或集体已经发表或撰写过的作品或成果。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本声明的法律结果由本人承担。

论文作者签名：李硕新 日期：2020年7月5日

## 学位论文使用授权说明

(必须装订在提交学校图书馆的印刷本)

本人完全了解北京大学关于收集、保存、使用学位论文的规定，即：

- 按照学校要求提交学位论文的印刷本和电子版；
- 学校有权保存学位论文的印刷本和电子版，并提供目录检索与阅览服务，在校园网上提供服务；
- 学校可以采用影印、缩印、数字化或其它复制手段保存论文；
- 因某种特殊原因需要延迟发布学位论文电子版，授权学校 ☐ 一年 / ☐ 两年 / ☐ 三年以后，在校园网上全文发布。

(保密论文在解密后遵守此规定)

论文作者签名：李硕新 导师签名：孙研  
日期：2020年7月5日

