计 算 机 研 究 与 发 展
Journal of Computer Research and Development
https://crad.ict.ac.cn
63(4): 1079−1093, 2026

面向服务器应用的远距离函数调用指令预取优化

陈 立[1,2] 高 军[1] 赵天磊[1] 刘 峤[3]
[1](飞腾信息技术有限公司处理器核研发部 长沙 410028)
[2](电子科技大学(深圳)高等研究院 广东深圳 518100)
[3](电子科技大学计算机科学与工程学院 成都 611731)
(chenli1577@phytium.com.cn)

Long-Distance Function Call Instruction Prefetching Optimization for Server Applications

Chen Li[1,2], Gao Jun[1], Zhao Tianlei[1], and Liu Qiao[3]
[1](*Processor Core Research and Development Department of Phytium Technology Co., Ltd., Changsha* 410028)
[2](*Shenzhen*) *Institute for Advanced Study*, *University of Electronic Science and Technology of China*, *Shenzhen*, *Guangdong* 518100)
[3](*School of Computer Science and Engineering*, *University of Electronic Science and Technology of China*, *Chengdu* 611731)

Abstract The large instruction fetch delay which is caused by the miss of L1 instruction cache is one of the most important bottlenecks that restrict the performance development of modern processors, especially in server applications which have a very large number of instruction traces. Instruction prefetching technology is a key technology to solve this problem, and it achieves the purpose of masking the high access latency by putting the instruction cache blocks to be used in L1 instruction cache in advance. In recent years, researchers have proposed a lot of instruction prefetching architectures to alleviate this problem, but due to poor temporal and spatial locality, long-distance function calls still bring a large number of missing instructions. In this paper, a new instruction prefetching mechanism is designed, which can achieve high coverage and high accuracy prefetching of the target instruction for function calls with low hardware overhead. Experiments show that after applying the optimization in this paper, the missing rate of the function call target instruction is about 45% lower than that of the current state-of-the-art instruction prefetcher, and the IPC (instruction per cycle) performance is about 11.9% higher than the baseline, and about 2.9% higher than the current state-of-the-art instruction prefetcher with similar overhead.

Key words instruction prefetching; server-side; function call; IPC; storage optimization

摘 要 一级指令缓存缺失导致的大取指延迟是制约现代处理器性能进一步提升的重要瓶颈之一,尤其在大指令踪迹的服务器应用上更是如此。指令预取技术是解决这一问题的关键性技术,它通过提前将要用到的指令块放入上级缓存中,从而达到掩盖高昂访问延迟时间的目的。近年来,研究者们提出了许多指令预取架构来缓解该问题,但由于指令局部性较差,长距离函数调用仍然带来了大量的指令缺失。设计了一种新的指令预取机制,能以较低的硬件开销实现对函数调用目标指令的高覆盖率和高准确率预取。实验表明,优化后的函数调用目标指令缺失率较目前最先进的指令预取器降低约45%,IPC(instruction per cycle)性能比基准线高约11.9%,比目前相似开销的最先进的指令预取器高出约2.9%。

关键词 指令预取;服务器端;函数调用;每周期指令数;存储优化

收稿日期:2024-10-10;**修回日期**:2025-10-10

基金项目:国家自然科学基金项目(U22B2061);国家重点研发计划项目(2022YFB4300603);四川省科技计划项目(2023YFG0151)
This work was supported by the National Natural Science Foundation of China (U22B2061), the National Key Research and Development Program of China (2022YFB4300603), and the Sichuan Science and Technology Program (2023YFG0151).

中图法分类号 TP302

DOI: 10.7544/issn1000-1239.202440783 CSTR: 32373.14.issn1000-1239.202440783

在过去数年里，服务器端程序的整体规模以飞快的速率不断膨胀。据统计[1]，服务器端指令足迹以每年约 27% 的幅度进行增长，而该幅度已远远大于如今一级指令缓存的容量增长速度，造成了极大的存储空间不足问题。服务器端的指令集大小以数十成百倍溢出一级指令缓存的存储大小，带来极其严重的缓存延迟以及性能损耗问题。除此之外，服务器端的深调用栈特性也为一级缓存带来了新的挑战。一个用户请求在通往服务器端路上，往往要经过网页服务器、自定义脚本、内核中的存储和网络 I/O 路径[1] 等指令足迹与软件栈。同时如今模块化编写特点也让代码分布愈加离散，函数调用频繁，为一级缓存的容量带来极大的压力，也容易造成更大的函数调用缺失率。函数调用表示程序的全局控制流往往有着更大的跳转距离，如在 C++中调用函数 *sort*() 需在标准库中找到指定头文件，它往往和当前主函数相隔甚远；而其他分支类型则表示程序的局部控制流，它们通常在一个函数内部，如条件分支“if…else”的目标指令通常在数十行代码内，空间局部性较好以致缺失率相对较低。在 Champsim 模拟器[2] 中，利用基准架构运行部分指令缺失率较大的服务器端程序，可以得到各分支类型目标地址的缺失情况，如表 1 所示。其中无条件分支指令中的直接调用(非直接调用)缺失率是最高的，超过了 54%。

Table 1 Target Address Missing Situation for Each Branch Type

表 1 各分支类型目标地址缺失情况

分支类型	总数量	缺失数量	缺失率/%
直接跳转	23 336	6 010	25.75
非直接跳转	7 948	2 559	32.20
条件分支	74 436	24 736	33.24
直接调用	33 077	18 166	54.92
非直接调用	6 043	3 708	61.36
返回指令	40 132	3 999	9.964

针对上述性能瓶颈问题，研究者们设计了多种指令预取结构来进行缓解。指令预取的含义是：在计算机多级缓存结构中某个缓存行(cache line)被程序访问前，通过某种算法以及程序流执行的某种规律提前将该缓存行从下级缓存中存入一级指令缓存(I-Cache)，以掩盖高昂的下级缓存访问延迟并提高处理器的执行效率。最早提出的是顺序指令预取器[3]，它的主要思想是当程序访问当前指令行时，下一行或几行的指令也会在不久被访问到，从而对下一行或下几行指令进行预取操作。该策略能有效应对顺序指令流的执行，然而却不能应对程序流中存在的跳转情况。当该类情况发生时，顺序预取器失效甚至会带来性能损耗。对此，20 世纪 90 年代 FDIP(fetch directed instruction prefetching)预取结构[4] 被提出，该结构的核心思想是利用分支预测器(branch predictor)的预测结果来作为指令预取的目标。应用该策略可以在程序流中存在跳转行为时还能给出准确的目标地址进行预取，弥补了顺序预取器的不足，且其硬件实现成本较低，因为其主要实现结构已经存在于处理器基本架构中。然而，FDIP 也存在着不足，它的预取准确度取决于分支预测正确率，可能会造成较多错误预取[5]；FDIP 结构的预取响应及时性也不够理想，可能会造成较多过晚预取(late prefetch)从而影响预取效果。为应对该问题，研究者们又研发了时间流(temporal streaming)预取器[5-8]。其原理是程序执行的时间相关性，即程序在一段时间内执行过的指令流序列很有可能在未来一段时间内得到相同指令流的复现。应用该结构可以大大提升预取的准确度以及及时性，弥补了 FDIP 结构的不足。但是时间流预取器的一大缺陷是需要大量的存储空间，而这一需求是现代处理器难以忍受的。在后续研究中[9] 又改善了 FDIP 结构，进一步引入了 BTB(branch target buffer)预取机制实现了更高的预取覆盖率和预测准确度。然而在应对指令集总数比较庞大的工作负载时，其 BTB 缺失率仍然较大，导致预取频繁失效。尤其在针对函数调用缺失时其缺失代价相对更大，直接导致 FDIP 结构整体失效[10]。由此提出了一类复合型预取器[10-11]，针对不同的缺失现象，引入“顺序+非连续+BTB”等预取结构，降低对 BTB 结构的耦合度，但是其也没有针对函数调用缺失进行优化。后续还提出许多预取结构，如 IPC1(the First Instruction Prefetching Championship)大赛中的一系列预取结构[12-19] 等。但是这些预取架构没有专门针对函数调用进行优化的结构，要么应对函数调用缺失比较乏力，要么就是需要大量的存储开销。而函数调用缺失问题在飞速发展的服务器端程序会愈加显著[20]，制约服务器性能的进一步发展。综上所述，本文认

为有必要引入针对函数调用缺失的指令预取器来弥补现有指令预取结构的空缺，同时呈现出和其他预取架构良好的正交性和叠加性。

针对上述问题，本文创新性提出 CTIP（call target instruction prefetch）预取结构，可以在较小硬件存储开销代价下，大大提高服务器端程序函数调用目标指令的覆盖率，且与其他预取结构呈现较好的正交性。本研究首先挖掘了函数调用指令的特性，利用该特性 2 个分离的表结构分别存储“签名-地址”的映射关系和完整目标地址信息，实现存储空间优化。其次，采用签名触发函数调用预取的方式，实现高精度预取的同时避免无效的存储开销。再者利用函数块的空间局部性，在预取函数目标地址的同时应用位向量策略预取其空间范围内的其余指令缓存块，也实现高效的存储压缩。应用该预取结构，可以让函数调用目标缺失率低至约 1.7%，远低于其他预取结构。同时结合 FDIP 结构的 CTIP 预取器可以达到最优的 IPC（instruction per cycle）性能，优于相似存储开销下目前最先进的指令预取器 EIP（entangling instruction prefetcher）[12]。本文的主要贡献有 3 点：

1）创新性引入类似 TAGE（tagged geometric history length branch prediction）[21] 的签名长度可动态调节预取机制来提高预取准确度。以往基于签名触发预取的指令预取机制都是基于定长的签名，而本文引入动态可调节签名长度的预取方式，在提高签名触发预取准确度的同时降低不必要的带宽开销和存储资源损耗。

2）利用调用指令函数的空间相关性特征，在对调用目标地址本身进行预取的同时创新性地引入位向量预取策略，对其函数块内的其余指令块也实现压缩预取，节约存储开销的同时提高预取覆盖率，较好地捕捉到函数块内部空间局部性特征并加以应用。

3）应用该预取结构可以达到目前最低的函数调用缺失率，仅有约 1.7%，比基准结构降低约 95.7%，比同等开销目前最优的预取结构[12] 降低约 45%；同时 CTIP 结构可以达到目前最优的预取器的 IPC 性能提升，比基准结构提升约 11.9%，较同等开销下目前最先进的指令预取器提升约 2.9%。

1 背景及相关研究工作

1.1 研究概况

指令预取（instruction prefetch）技术自 20 世纪 90 年代发展起来[3]，如今已经成为计算机体系结构中不可或缺的关键性技术之一。由于计算机体系结构中一级指令缓存空间过小不能满足现代指令集的需求，从而造成大量的缓存缺失，大大限制处理器性能的进一步提升，因此指令预取技术应运而生。在近些年中，指令集朝着越来越庞大的方向发展。据统计[1]，每年指令集的平均增长幅度约为 27% 并朝着增幅越来越快的方向增长。与此同时，服务器端指令集日渐呈现出深调用栈的特点[22]，这让较高的函数调用目标缺失率成为新的问题。传统预取结构多用于处理空间跳转范围较近的指令，如顺序预取器[3]、步幅预取器等。而 FDIP 预取器[4] 则受限于分支预测准确率和 BTB 容量大小问题，在应对函数调用目标指令缺失时较为乏力。另一类预取器——时间流预取器[5-8]，虽然能有效应对全局流跳转行为，但是其所需的存储开销（metadata）太大，在实际结构中难以实现。由此可见，设计出存储开销友好型，并针对函数调用目标缺失的预取器是有较大价值的。在未来服务器端应用指令总数以及调用栈深度进一步提高时，该结构存在的必要性将被进一步体现。

1.2 FDIP 预取结构

20 世纪 90 年代发展出 FDIP 预取结构[4,23-24]。其实现的核心思想是：利用分支预测器提供关于未来指令流的预测信息来指导指令预取。FDIP 很好地弥补了顺序预取器无法应对程序控制流跳转行为的问题。分支预测是计算机体系结构另一关键技术，其主要目的是通过学习程序过去一段时间内的执行情况来动态预测未来控制流的动向，应对程序中存在的分支跳转问题，让取址单元可以源源不断地得到取址信息。利用分支预测的输出作为指令预取的预取目标，并将其缓存在一个独立于流水线的解耦结构 FTQ（fetch target queue）中，从而达到可以源源不断为主流水线提供预测指令流信息的效果。然而，这类预取器也存在其局限性。首先，预取的成功率大大依赖于预测的准确性，一旦预测发生错误，就会造成错误预取，缓存污染与流水线冲刷，造成性能损耗。而且一条错误的预测往往会跟随一系列的错误预测，其造成的性能损耗也是不容小觑的。此外，FDIP 预取的及时性也是一个问题[5]，其往往需要连续正确预测多条分支才能达到预取前瞻性的要求，但是这也为分支预测的准确性提出了很严苛的要求。该效应在应对函数调用时尤为显著，因为要正确预取一条调用目标指令就需将其前面一系列条件分支链都预测正确，此时预测正确率已经降到很低了。同时此时 BTB 中相应分支的缺失率也较大，进一步抑制其

在预取函数调用指令中的表现。综上所述,FDIP结构在应对服务器端调用密集型程序中表现得不尽如人意。这也是本文想引入函数调用指令优化的原因之一。

1.3 时间流预取

时间流预取器[5-8]在FDIP结构被提出后逐渐发展起来。该结构总结了FDIP结构的不足,认为预取器的优化主要有2个方面:一是FDIP顾及不到的预取前瞻性(及时性)问题;二是预取的准确性问题,它认为FDIP的准确性还达不到要求。针对该出发点,研究者们研发了新的预取结构,利用指令流的"时间相关性",也就是"记录并重现(record and replay)"的思想,通过记录一段完整的指令流时间序列,并在程序下一次执行到相同位置时依次预取该流后续的指令序列,达到较高的预取准确率和及时性。与FDIP结构不同,它并不依赖于分支预测的准确率,而只是依赖于该结构本身存储的元数据(metadata)。通过记录指令流的缺失序列(miss sequence)[5]或访问序列(access sequence)[6],可以达到非常高的预取准确率,且由于其记录的是完整的时间流序列,预取及时性问题也得到了很好的解决。但时间流预取也存在明显弊端,完整记录指令的时间流序列需要大量的存储开销,尽管后续一部分优化方案被提出[7-8,25],总存储开销还是达到了令人望而生畏的地步。究其根本原因是存在许多冗余开销,如一小段时间流序列可能在整个程序流序列中重复存储多次,而实际上只需对相同模式进行1次存储即可。其次对于顺序执行的缓存块,也可以有更好的方案来进行压缩存储。本文也针对时间流预取的弊端进行了优化,实现了更高的存储效率。

1.4 其他预取结构

除了上述的预取结构外,还有许多各式各样的预取结构。RDIP(return-address-stack directed instruction prefetching)[25]结构在上述时间流结构的基础上,引入"签名触发预取"的策略。其利用RAS(return address stack)栈中存储的调用信息来指导触发相应的预取;D_JOLT[14]在此基础上引入新的签名生成结构,确保用来生成签名的元素是最新的函数块以提高预取准确率。Boomerang[9]结构在传统FDIP结构上添加BTB预取机制,一定程度上缓解了BTB缺失率较大导致预测正确率较低及预取效果不佳的问题;Shotgun[1]一文在Boomerang的基础上进一步改良了BTB结构,将其分为U-BTB(unconditional BTB)和C-BTB(conditional BTB)两个部分。其中U-BTB用于存储函数调用指令;C-BTB用于存储条件分支指令,并将较大比例存储空间分配给U-BTB,同时利用BTB预取策略实时更新C-BTB的表项,从而达到更高的BTB覆盖率,改进FDIP的预取效率。这一措施也侧面印证了函数调用缺失对于系统整体性能的影响是比较大的。上述预取器都有一些显著的特征,那就是它们都会以函数调用指令作为全局控制流来指导触发预取。尽管实现的结构各不相同,但是都反映了一个关键信息:用全局流信息来指导预取具有较高的可靠度以及空间压缩的能力。除此之外,还有其他不同预取策略的预取器,如Divide and Conquer[10]应用了"分而治之"的思想,将缺失分成顺序缺失、非连续缺失以及BTB缺失3种情况,应对不同缺失采用针对性预取策略,取得较好的效果;EIP[12,26-27]预取器着眼于预取的及时性问题,提出"不同指令缺失应该有不同访问延迟",从而针对性地设定预取距离,并赢得了IPC1大赛的头筹。还有一类预取器[28-31]有机结合FDIP结构与专用预取器,通过合理的资源分配实现在较低硬件开销下达到更优的性能提升。除此之外,文献[32−34]提出用配置文件指导触发软件预取,带来性能收益的同时减少相应的硬件存储开销。

但是总结来说,目前没有一款预取器是基于解决函数调用目标缺失进行设计的。虽然函数调用只占分支总数的20%左右,但是其缺失率却是最高的(如表1所示),且其缺失代价也比较高。随着服务器端指令总数不断增长以及调用栈深度不断提高,调用目标缺失问题会日渐严峻,因此本文提出了针对函数调用目标缺失的优化策略。此外,由于函数块内部有着较好的空间局部性,对函数调用目标地址的成功预取也有利于对其空间区域内其他指令块的预取,从而实现高效预取存储压缩。

2 实现结构

2.1 实现思想与原理

由于全局调用目标地址的空间局部性较差,导致其在所有指令类型中缺失率最高。文献[35]也表明减少调用缺失与减少顺序缺失或条件分支缺失具有同等重要性。同时由于目前没有有效的应对调用目标地址缺失的预取方法,引入调用目标地址的预取算法就显得尤为关键。如何才能实现对其高效且准确的预取呢?实验调查发现,在服务器端程序中的函数调用行为特别频繁,但是其包含的函数总数

却相对较少。这就表明，同一段函数可能在程序的不同区域被调用多次，而且随着程序的扩大，调用次数会不断增加，而被调用的函数总数却不会明显增长。如表 2 所示，在 Champsim 模拟器中的基准架构下运行不同长度的同一程序段，随着程序规模的不断扩大，每个函数被调用的次数提升明显，但函数总数却无明显变化，每个函数被调用的平均次数也大幅提升。这就表明，像时间流预取般的存储策略必然会存在大量的冗余数据，如同一个函数目标地址可能出现在时间循环队列的多处。

Table 2 Number of Function Calls Under Different Program Sizes

表 2 不同程序大小下函数调用次数

函数调用次数	1×10^6	5×10^6	10×10^6	20×10^6	50×10^6
调用总次数	35 911	177 152	352 844	704 807	1 745 559
函数个数	866	918	918	966	1 292
平均调用次数	41.47	193.0	384.4	729.6	1 351.1

如何实现较高预取准确率的同时保证较低存储开销成为一个关键问题。实验表明，尽管同一个函数被调用的次数可能很多，但是在所有调用情况中，不同的调用环境(context)往往只有少数个。如某函数在一段程序内一共被调用 600 次，其中包含了 3 种不同的调用环境，环境 1 调用 100 次，环境 2 调用 300 次，环境 3 调用 200 次。只需完整记录不同的调用环境而不是所有的调用次数，就可以较好地概括该函数的所有调用情况。而这段环境可以由该函数被调用前的一部分全局控制流历史来很好地概括。如图 1 所示，对于当前调用函数 X 来说，预取距离前的一段全局流历史(函数调用)序列为 $ABCDE$，而这段历史调用序列便是对应调用函数 X 的调用环境。在下次遇到相同调用环境时，即可触发对目标函数 X 的预取，从而实现较高的预取准确率以及覆盖率。文献 [36] 同时表明全局控制流的预测精度相对更高。由此，用不同的结构分别存储函数目标地址及其调用环境的对应关系，就可以实现在较小存储开销下达到较高的预取精确度，实现对函数调用目标指令及其所在函数块空间区域的预取。

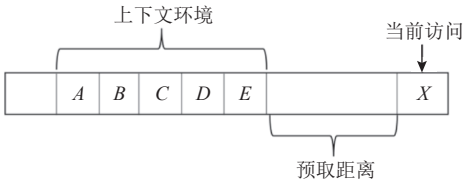


Fig. 1 Relationship diagram of function calls and their invocation environment

图 1 函数调用及其调用环境关系图

如上所述，图 1 结构类似于用签名触发相应预取目标的预取策略。而签名长度也是影响该类预取策略准确性的关键性因素。过长的签名会导致噪声的引入，从而造成存储开销的浪费以及学习周期的增长；而过短的签名又会造成预取准确度降低，带来无效预取、带宽压力以及缓存污染。由此可见，适时调整合适的签名长度对于预取效率来说是很关键的。而目前在指令预取领域，还未引入签名长度动态可变的预取策略。如文献 [25] 中采用一种固定长度的签名生成方法；文献 [14] 引入 2 种不同长度的签名，分别对应长距离预取器以及短距离预取器，但其签名长度的设定仍是静态化的。通过在不同调用情况下引入不同的签名长度并对其进行动态调整，可以避免生成过长或过短的预取签名，达到资源优化和预取效果的最佳权衡。

本文借鉴了 TAGE 分支预测器[21]，为指令预取引入针对函数调用指令的签名长度来动态调整预取结构。其签名长度调整的原理为：设定签名初始长度为 4，在检测到预取失效(prefetch miss)时(即在最近预取表中找不到对应目标地址)则说明要分配新表项，分配策略如图 2 所示。其中预取失效分为 2 种情况：没有发出预取(no prefetch)以及发出了错误的预取(wrong prefetch)。其中前者只需分配初始长度为 4 的签名即可，而后者则需在错误预取的基础上生成更长的签名以提高区分度。如错误预取对应的签名长度为 8，则此时需分配的签名长度为 16。在触发预

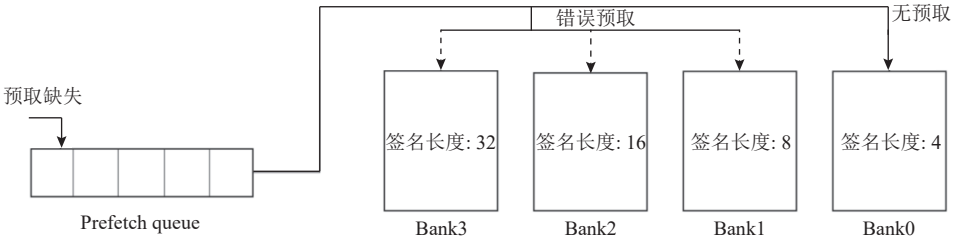


Fig. 2 Allocation strategy of signature length when prefetch is missing

图 2 预取失效时签名长度分配策略

取时，优先触发签名长度更长的预取，因为此时其蕴含的历史信息量是更准确的。由此一来，对于签名冲突较少的项，它们的签名长度就能被维持在比较小的水平；而对于频繁发生签名冲突的项，就会不断往上分配签名长度，直至能把它们清晰分辨出来为止。由此便完成了对签名长度的动态调整策略，以适应不同调用情况的需求。

除此之外，程序控制流可以分成全局控制流和局部控制流。其中，全局控制流由函数调用及其返回指令所控制，局部控制流则是由函数内部的条件分支等指令所控制，它们两者的关系如图 3 所示。

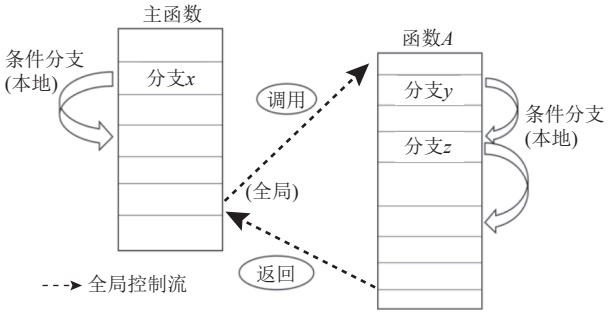


Fig. 3 Relationship between global control flow and local control flow

图 3 全局控制流与局部控制流关系

其中，全局控制流有着较大的跳转距离，而局部控制流则有着相对较小的跳转距离(因为其只存在于函数内部，而一个函数的大小往往在数个缓存块以内)。如图 4 所示，85% 左右的函数块大小都在 8 个缓存块以内。由此可见，通过触发地址加位向量的存储模式，可以比较好地刻画出一个函数的整体调用情况。其中触发地址为调用目标地址，而位向量则是由一系列 0，1 位组成的数组，整体反映了函数内部调用情况。如 0001001 则反映了该函数内部第 4 个和第 7 个缓存块是会被使用的，从而需要触发相应的预取，位向量模式如图 5 所示。利用这种存储方式，可以用较小存储开销比较完整地反映某个函数块内部的使用情况，很好地利用了函数块内部的空间局部性来实现压缩存储。

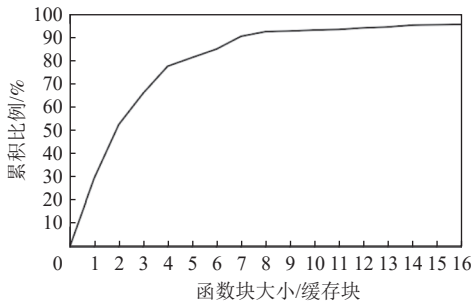


Fig. 4 Cumulative distribution diagram of function block size

图 4 函数块大小累积分布图

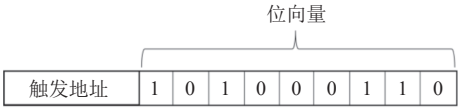


Fig. 5 Schematic diagram of the bit-vector mode

图 5 位向量模式示意图

2.2 实现具体结构

图 6~8 为 CTIP 预取结构的整体示意图(预取训练/触发)。其中 sig table 用于存储签名和目标地址索引的对应关系；target table 用于存储调用目标地址及其所在空间区域的位向量；extra table 用于存储部分超出位向量范围的函数块及其额外位向量用作补充；prefetch queue 用于存储最近触发过的预取的目标地址及其预取签名，用来辅助预取训练以及触发；history queue 用于存储过去一段时间程序的全局调用

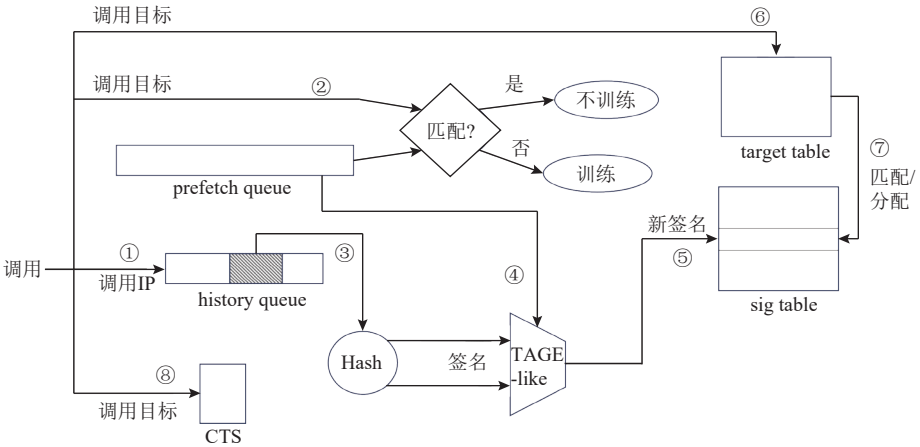


Fig. 6 Prefetch training process of call instructions

图 6 调用指令的预取训练过程

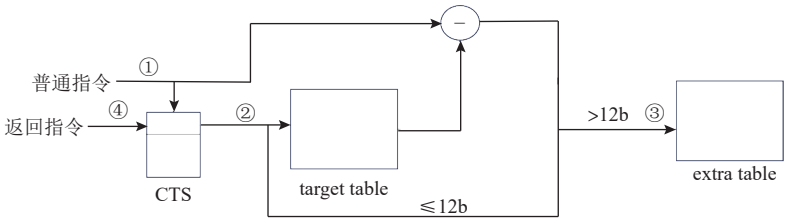


Fig. 7 Prefetch training process for normal instructions and return instructions

图 7 普通指令与返回指令的预取训练过程

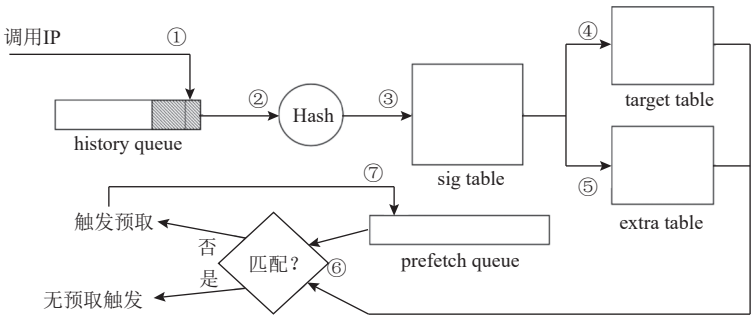


Fig. 8 Prefetch triggering process for call instructions

图 8 调用指令的预取触发过程

情况(调用指令 IP 地址)来指导生成预取信息以及触发预取; Hash 是一个功能模块，用于生成以 history queue 中存储的信息为原材料的签名并用此来访问 sig table; CTS(call target stack)是一个类似于 RAS 的结构，但是它存储的是函数目标地址而不是返回地址，作用是更新相应函数块的位向量。此外，CTIP 架构还会结合 FDIP 预取以实现对局部控制流预取的补充。

2.2.1 预取训练

如图 6 与图 7 所示为 CTIP 结构进行预取训练的过程。其中，根据不同指令类型训练分成 2 个部分：针对调用(call)指令的预取训练；针对返回(return)以及其余指令(normal)的训练。其中，调用指令的训练为 CTIP 结构的主要部分。

图 6 为调用指令的预取训练过程。将调用指令的 IP 地址存入 history queue 这个 FIFO 结构中，用于生成后续的预取关联对(①)。用调用指令的目标地址查询 prefetch queue，如在该队列中找到目标地址，说明该地址在最近被正确预取过，无需再生成训练。如果 prefetch queue 内没有找到该地址，说明此时有必要生成预取对(②)。从 history queue 中取出若干项历史信息用于生成签名来关联当下目标地址。这些历史信息通过哈希算法生成一系列长度不一的签名(③)。将其再次与 prefetch queue 的表项进行匹配(具体原理在 2.1 节中进行了解释)，决定最终要分配的签名长度(④)。sig table 以签名生成的索引(index)和标签(tag)进行匹配，如无匹配则分配新项至表中，采用 LRU 置换策略(⑤)。调用指令同时还需搜索其目标地址是否存于 target table 中，如不存在则需分配一个新项到表中(⑥)。同时返回分配或匹配项对应的索引到 sig table 对应项中，用于生成从签名到地址信息的对应关系(⑦)。还需将该调用指令目标地址压栈到 CTS 中，以指示程序当前执行到哪个函数块，用于后续相应函数位向量的生成(⑧)。

图 7 为普通(非调用或返回)指令和返回指令的预取训练过程。其中：普通指令首先访问 CTS 栈顶，确定程序此时所属的函数块是哪一个(①)。利用该信息搜索 target table，找到对应项，并通过计算该普通指令地址与该函数目标地址的差值来将对应的位向量置 1，如两者的地址差为 8，则将第 8 位位向量置 1，以此类推(②)。倘若两者的差值大于位向量能表示的最大范围(12b)，将位向量信息填入 extra table 中。如两者差值为 18，则将 extra table 中对应项的第 6 位位向量置 1，以此类推(③)。当访问指令为返回指令时，需在 CTS 栈上执行弹栈操作，确保此时 CTS 栈反映的函数调用信息是准确的，程序从子函数返回父函数(④)。

2.2.2 预取触发

图 8 为 CTIP 结构触发预取的过程。该结构主要由调用指令触发相应的预取操作。将新的调用指令 IP 地址存入 history queue 中(①)，结合队列内之前的历史信息一起生成新的签名(②)以触发预取。通过

哈希算法得到一系列签名后(③)，以签名长度由长到短及优先度依次递减的顺序匹配 sig table 的表项。如同时有签名长度为 8 和 4 的项匹配了 sig table，则此时只会触发长度为 8 对应的预取，因为此时该项对应的信息是更准确全面的。完成 sig table 匹配后，会得到关于目标地址的索引，此时便用该索引分别访问 target table(④)和 extra table(⑤)，找出对应项信息，包括预取目标的完整地址以及其对应要触发的位向量等。在生成预取请求前(⑥)，还需将相应的目标地址与 prefetch queue 中的项进行匹配，只有当都不匹配时才触发该预取，以保证该项在最近一段时间内没有被预取过，以防止冗余预取。还需将触发过的预取信息(包括签名和目标地址信息)填入 prefetch queue 中(⑦)，以便后续的预取请求进行预取历史查询。以上就是 CTIP 结构整体预取触发的过程。

3 实验方法

本文采用 Champsim[2] 模拟器架构进行实验和数据评估。Champsim 模拟器是一个基于 trace 的模拟器，被广泛应用于学习和赛事中，DPC3(the Third Data Prefetching Championship)以及 IPC1 等赛事都采用了该模拟器，说明该模拟器具有较高的权威性和泛用性，其得到的实验数据也比较有信服力。本文采用了在原 Champsim 结构上进行修订后的版本[37]，在原有基础上引入了最新的 FDIP 结构，同时对原有的 BTB 结构进行改良，引入更真实且可配置的 BTB 文件，比原架构更接近真实处理器的表现。模拟器各参数设置如表 3 所示。

本实验所采用的工作负载为 DPC3 以及 IPC1 大赛所使用的负载，这其中包括 client 程序、server 程序以及一部分 SPEC 标准测试程序。由于本文主要针对的是服务器端深调用栈程序的优化，所以会主要分析其中 server 程序的实验数据，因为通常它们的函数调用缺失率问题比较严重。每道程序都会首先用 50×10^6 指令预热(warmup)，再用后续 50×10^6 指令来收集数据(evaluation)，该配置也是和大多数有关指令预取结构的论文中保持一致[12-19]。

本文主要用到的指令预取对照结构，包括基准无预取结构(no_pref)、顺序预取结构(next_line)、FDIP 预取结构[38-39]、CTIP 预取结构、EIP128(entangled instruction prefetcher 128 KB)预取结构[12] 以及 EIP35(entangled instruction prefetcher 35 KB)预取结构。其中 EIP 预取结构赢得了 IPC1 大赛的冠军头衔，也是目前最先进的指令预取结构之一。

Table 3 Simulator Configuration Parameters

表 3 模拟器配置参数

缓存体系	配置	参数
CPU	频率/GHz，核数量	4，1
	BTB 行数	4 096
	预测错误惩罚/cycle	1
	分支预测器	Hashed perceptron
	ROB 行数	352
	LQ 行数及 SQ 行数	128，72
	RAS 行数	64
	取指宽度及译码宽度	6，6
	执行宽度及退休宽度	6，4
	LQ 宽度及 SQ 宽度	2，2
	Scheduler 行数	128
I-Cache	缓存行数及 ITLB 行数	64×8，16×4
	RQ 行数及 WQ 行数	64，64
	PQ 行数及 MSHR 行数	32，8
	访存延迟/cycle	4
	缓存行长度/B	64
D-Cache	缓存行数及 ITLB 行数	64×12，16×4
	RQ 行数及 WQ 行数	64，64
	PQ 行数及 MSHR 行数	8，16
	访存延迟/cycle	5
	预取器	Next-line prefetch
L2-Cache	缓存行数	1 024×8
	RQ 行数及 WQ 行数	32，32
	PQ 行数及 MSHR 行数	16，32
	访存延迟/cycle	10
	预取器	Spp-dev
LLC	缓存行数	2048×16
	RQ 行数及 WQ 行数	48，48
	PQ 行数及 MSHR 行数	32，64
	访存延迟/cycle	20
	预取器及置换策略	No，LRU

本实验主要收集的实验数据包括 IPC 性能、函数调用缺失率(call miss rate)、系统的预取带宽(prefetch bandwidth)开销以及预取准确率(prefetch accuracy)。具体实验数据会在第 4 节进行对比展示。

4 结果与数据分析

4.1 存储开销

CTIP 一共用到了 6 种不同的存储结构，它们所

占开销大小如表 4 所示，总共的存储开销约为 32.9 KB。其中，sig table 和 target table 是 CTIP 的重要组成部分，也占用了较大存储开销，所以对其的存储优化也是有必要的。本结构引入了“sig table + target table”的分层存储结构，减少了整体存储开销约 30%。实验发现，当 sig table 取 512×8 组关联时，能取得较好的性能和存储开销间的权衡。

Table 4 Storage Overhead of the Prefetch Structure

表 4 预取结构存储开销

存储结构	每行开销/b	总行数	存储开销/KB
sig table	39	512×8	19.5
target table	70	64×20	10.94
extra table	90	128	1.41
history table	58	64	0.45
prefetch queue	82	16	0.16
CTS	58	64	0.45
总存储开销			32.91

sig table 为 512×8 组相联结构，每行用 21b 存储 target table 对应项的索引。target table 为 64× 20 组相联结构，每行存储 58 b 完整目标地址以及 12 b 位向量。分离 2 个表结构的目的是避免重复存储目标地址及其位向量，以防造成无用的存储开销。如果只用单一表结构进行存储，则相应的存储开销为 4 096×88，即 44 KB，而分离表结构后只需约 30.5 KB，降低约 30%。在目前许多存储结构中都可以采用类似策略来减少存储开销，如 BTB 中可能会存在较多目标地址重复的问题；EIP 中的 entangled table 虽然采用了压缩存储技术，但还是难以避免存在较多冗余目标地址的存储。该情况在函数调用指令中尤其明显，因为其对应的目标函数（目标地址）较少而重复调用次数较多，所以应用该策略得到的优化程度较大。

与其他预取结构相比，CTIP 有显著的存储优势。EIP[12] 结构约消耗 128 KB 存储空间，PIF[6] 约消耗 300 KB 存储空间，D_JOLT[14] 约消耗 125 KB 存储空间。CTIP 实现了在较小存储空间开销下对函数调用高达 98% 以上的覆盖率，远高于上述其余预取器。可见该预取结构在应对函数调用缺失时的优势是比较巨大的。

4.2 函数调用缺失率

本节主要考察的是 no_pref, FDIP, CTIP, EIP35, EIP128 预取结构的函数调用缺失率变化，主要对比的是服务器端应用程序，因为这类程序中函数调用缺失问题比较严重，具体数据如图 9 所示。

由图 9 可知，在 server 端程序中，函数调用缺失问题还是比较严重的。基准结构在 server_17~server_22 以及 server_36~server_39 程序中接近甚至超过 50% 的缺失率，平均函数调用缺失率达到 39.67%，如表 5 所示（其中平均数据采用几何平均进行计算）。而在应用 FDIP 预取结构后，该情况也未能得到有效缓解，其平均缺失率也高达约 34.7%。这一数据也体现了应对函数调用缺失的重要性，因为在 FDIP 结构中仍有大量的函数调用缺失未被覆盖。在 EIP35 结构中，函数调用缺失率平均降至约 3.1%，但是在部分调用密集型程序中，如 server_03~server_13 以及 server_23~server_27 中，函数调用缺失率还是高达 10% 左右，限制程序性能的进一步提升。相比之下，CTIP 结构的函数调用缺失率表现更好，平均只有 1.7% 左右，且在上述部分缺失率较高的程序中，其函数调用缺失率也得到了有效控制，维持在 2% 左右。由此可见，

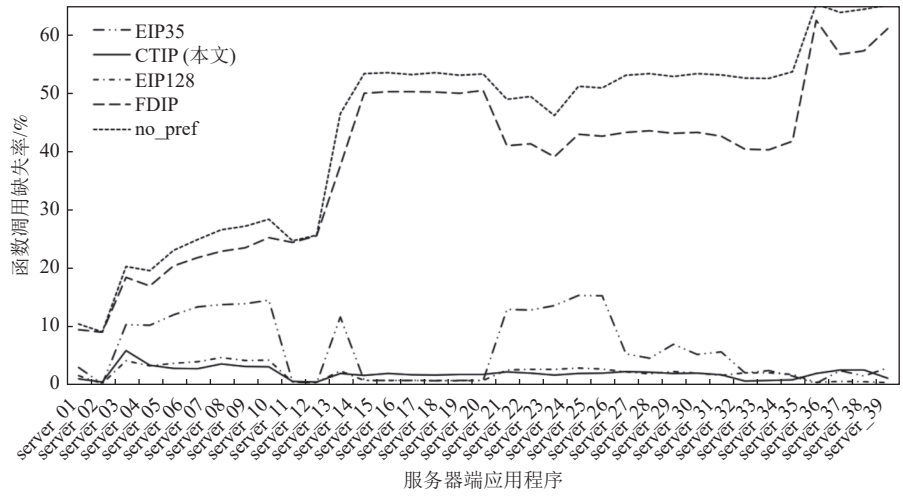


Fig. 9 The missing rates of function call of each prefetch structure in server-side applications

图 9 各预取结构在服务器端应用中的函数调用缺失率

Table 5 Average Missing Rates of Function Call for Each Prefetch Structure

表 5 各预取结构的函数调用平均缺失率 %

预取结构	基准	FDIP	EIP35	CTIP（本文）	EIP128
缺失率	39.67	34.74	3.13	1.71	1.43

在相似存储开销下，CTIP 结构在应对函数调用时有较大优势，缺失率较 EIP35 减少约 45.37 个百分点。与存储开销是 CTIP 约 3.9 倍的 EIP128 相比基本持平。相较 FDIP 结构，函数调用缺失率降低约 95.1 个百分点。由此可见，CTIP 结构可以很好地弥补 FDIP 结构在应对函数调用时乏力的问题，同时可以避免造成过大的存储开销，在性能和存储开销间取得了很好的权衡。且在图 9 中可以发现，在程序上函数调用缺失问题越严重，CTIP 结构的优势越明显，如在 server_03~ server_13 以及 server_23~server_27 程序段，这也侧面印证了 CTIP 结构在应对调用密集型服务器端程序时的应用潜力巨大。

4.3 IPC 性能

本节主要考察的是 no_pref, next_line, FDIP, CTIP, EIP35, EIP128 预取结构的 IPC 性能差异，主要对比的是 server 应用程序中的性能变化，因为这类程序受函数调用缺失影响较大。各预取结构 IPC 性能对比如图 10 所示。

由图 10 可见，在 server 应用程序中，指令预取提升 IPC 性能的潜力还是很大的。在部分程序中如 server_12 以及 server_23~server_27, IPC 性能提升最高可达约 28%。各预取结构平均 IPC 性能表现如表 6 所示，采用几何平均计算方法计算平均数据。其中，CTIP 的平均 IPC 性能约为 0.86，基准结构的平均 IPC 性能约为 0.77，提升约 11.9%；FDIP 结构平均 IPC 性能约为 0.83，CTIP 较之提升约 3.6%；目前最先进的指令预取器 EIP35[12] 的平均 IPC 性能约为 0.84，CTIP 较之提升约 2.1%。

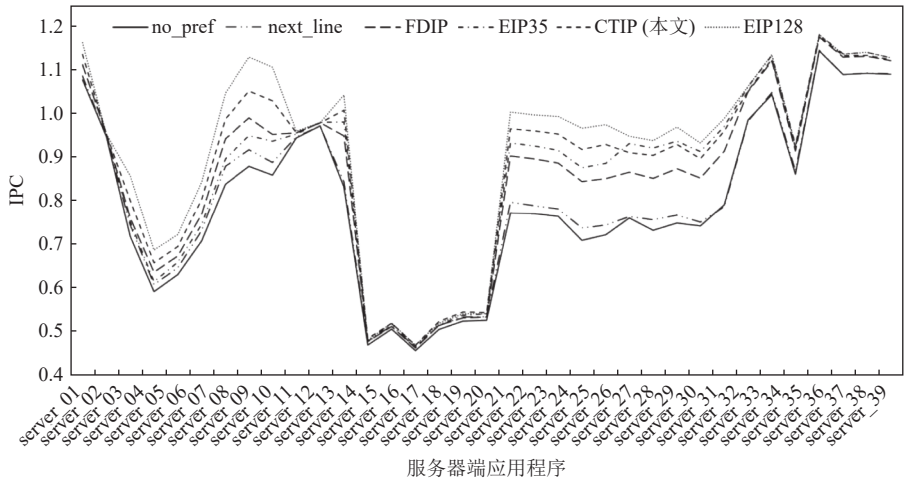


Fig. 10 IPC performance changes of each prefetch structure in server-side applications

图 10 各预取结构在服务器端应用中的 IPC 性能变化

Table 6 Average IPC Performance of Each Prefetch Structure

表 6 各预取结构的 IPC 平均性能

预取结构	基准	next_line	FDIP	EIP35	CTIP（本文）	EIP128
平均 IPC	0.768 5	0.781 7	0.830 9	0.842 0	0.860 0	0.881 9

进一步分析可以得到，在 server_03~server_13 以及 server_23~server_31 附近 IPC 性能提升较为明显。结合图 9 中的数据可以发现，这 2 个区间内的函数调用缺失率也是比较大的。如在 server_26 程序中，基准结构的函数调用缺失率约为 51%，FDIP 结构的函数调用缺失率约为 43%，EIP35 的缺失率约为 15%，而 CTIP 的缺失率仅有约 2%。CTIP 在该程序上对应上述结构的 IPC 性能提升分别约为 29.5%(no_pref)，8.8%(FDIP)，4.8%(EIP35)。这一数据说明了在频繁发生函数调用缺失的程序中，减少函数调用缺失率可以带来有效的 IPC 提升。这一规律在其余程序中也得到了印证。如 CTIP 结构对比 EIP35 结构，在 server_3~server_13 和 server_16 以及 server_23~server_27 中 IPC 提升较为明显，平均提升约 6%。而在这些区间内 CTIP 对 EIP35 函数调用缺失率的优化也是最为明显的，缺失率平均降低了约 80%。这一结论也进一步印证了在服务器端优化函数调用缺失率对 IPC 性能提升的巨大意义。

然而，仍有部分程序需要进一步解释。如表 7 所

Table 7 IPC Performance Comparison Between the Benchmark Architecture and the Perfect (No Missing) Instruction Cache Architecture

表 7 基准结构与完美（无缺失）指令缓存结构 IPC 性能对比

结构	服务器端应用程序									
	server_02	server_14	server_15	server_17	server_18	server_19	server_20	server_21	server_22	server_36
基准	0.946 5	0.943 1	0.970 3	0.468 3	0.504 2	0.455 1	0.504 2	0.523 0	0.524 8	1.142 3
完美	0.954 6	0.966 4	0.980 8	0.476 5	0.514 9	0.464 4	0.516 5	0.534 2	0.536 0	1.187 5
提升比/%	0.86	2.46	1.08	1.75	2.12	2.05	2.44	2.15	2.12	3.95

示，部分服务器端程序是指令缓存不敏感型的，体现在基准结构（no_pref）和完美指令缓存（perfect）的IPC差异上。在这些程序中两者的IPC性能差异平均为2%左右，说明在这部分程序上通过指令预取减少指令缓存缺失数无法带来明显的IPC提升。这解释了图9中相应程序段的函数调用缺失率优化明显却无法在图10中相应带来显著IPC提升的原因。剔除这一部分对指令缓存不敏感的程序来看，整体的IPC提升还是符合函数调用缺失率降低的变化趋势的。剥离这些特殊程序段后，CTIP结构比起FDIP结构IPC提升约4.7%，比起EIP35提升约2.9%。

此外，本文还对CTIP结构在整个存储体系中所占容量进一步对比，以揭示该预取结构的实用价值。如上文所述，CTIP预取器的总存储开销约为32.9 KB，而基准架构中I-Cache的总存储容量为32 KB。将基准架构的I-Cache容量分别扩容至64 KB/128 KB，并与原32 KB+CTIP结构进行对比，以验证如果将相似开销面积叠加到I-Cache中，得到的基准架构是否能取得与同等存储开销的CTIP结构相当的IPC性能提升。实验结果如表8所示，其中IPC采用几何平均方法计算。

Table 8 Performance Comparison of Baseline Architecture Under Differrent I-Cache Capacities and CTIP

表 8 不同 I-Cache 容量下基准架构与 CTIP 的性能对比

结构	64K	128K-I	128K-II	CTIP（本文）
IPC	0.772 0	0.781 9	0.782 0	0.860 0

注：CTIP表示32KB I-Cache + CTIP预取结构，64K表示128组8路组相联I-Cache扩容后的基准架构，128K-I表示256组8路组相联I-Cache扩容后的基准架构，128K-II表示128组16路组相联I-Cache扩容后的基准架构。

如表8所示，在相似存储开销下，CTIP（64.9KB）比起64K有约11.4%的IPC性能提升；比起存储开销翻了1倍的128K-I/128K-II，CTIP结构也有约10%的IPC性能提升。综上所述，在整个存储体系中加入CTIP结构会带来明显的性能提升，具有优秀的实用价值。

总体来看，CTIP结构的IPC比基准结构提升了约11.9%；比FDIP结构性能提升了3.5%~4.7%；比EIP35结构性能提升了2.1%~2.9%。结合函数调用缺失率来看，函数调用缺失优化较大的程序的IPC提升也较大，对指令缓存不敏感的部分程序除外。由此可见，CTIP在应用于服务器端等函数调用目标缺失率较大的应用程序获得的收益前景还是比较可观的。函数调用目标缺失也越来越成为抑制当今处理器在服务器端调用密集型程序环境下IPC性能进一步提高的关键因素之一。

4.4 带宽开销

图11为EIP35, CTIP, EIP128这3种预取结构发出的预取请求数即预取带宽开销对比图。可以看出，CTIP结构造成的预取带宽开销比EIP35以及EIP128都减少较多，其平均带宽开销比EIP35减少约61.2%，比EIP128减少约73.9%。在部分服务器程序如server_15中的优化程度更是达到接近84%。这是因为在EIP预取结构中，发出的大部分预取请求都是已经存在于指令缓存中的，造成了不必要的带宽开销。如server_02程序中，EIP35发出的预取请求，有超过95%本就存在于指令缓存（prefetch hit）中。而平均来看，EIP35发出的预取请求中，有约90%以上属于该类情况，造成大量无用的预取带宽开销。造成这一现象的原因是EIP策略预取的许多缓存块时间和空间相关性较好，它们在指令缓存中的缺失率较低，导致对这些缓存块的预取效益下降。相对来说，函数调用指令的空间和时间局部性更差，在指令缓存中的缺失率比其他指令更高，针对这类指令发出的预取有用的可能性也相对更大。所以在相似IPC性能表现下，其造成的相应的预取带宽开销会更小，而CTIP结构正是针对函数调用目标发出预取请求，所以其预取带宽开销相对较小。

此外，在部分函数调用目标缺失率较大的服务器程序如server_25中，CTIP带宽优化程度相对地较小，如server_25中比EIP35只降低了约35%。这是因

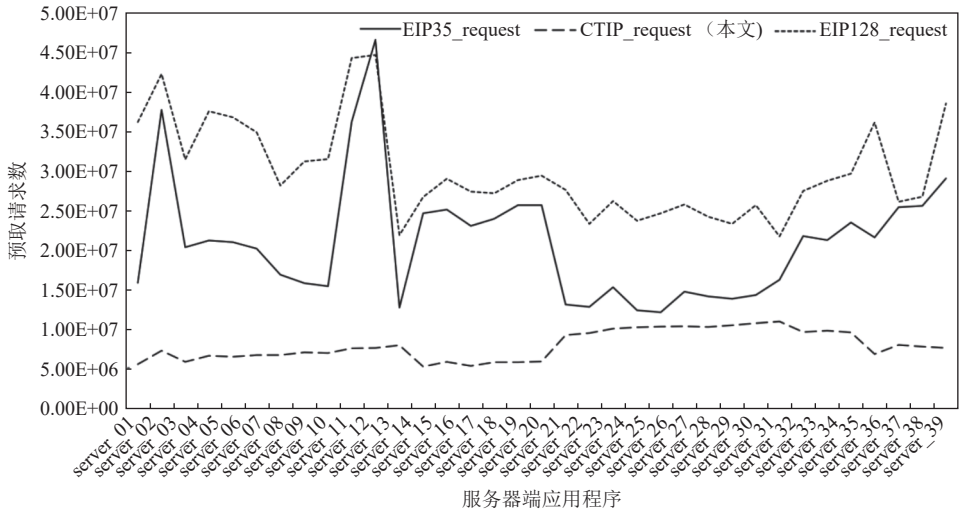


Fig. 11 Number of prefetch requests issued by each prefetch structure (prefetch bandwidth overhead)

图 11 各预取结构发出的预取请求数（预取带宽开销）

为在这类程序中函数调用目标缺失率较大，CTIP 发出的函数调用预取请求较多，相应带来的 IPC 提升以及函数调用覆盖率提升也是较为明显的。由此可知 CTIP 结构是函数调用缺失敏感型的，这一规律在剔除部分对指令缓存容量不敏感的程序后是普遍成立的。

总而言之，在同等 IPC 性能提升下造成更小的预取带宽开销就意味着相应发出的预取要更“高效”。实现更“高效”的预取，就意味着预取目标更不可能存在于指令缓存中，即预取缺失率更高。而在所有指令类型中，函数调用目标的缺失率是最高的。这表明在同等带宽开销限制下，针对函数调用指令发出的预取效益是最高的，带来的 IPC 提升也是最大的。如 CTIP 结构相比 EIP35 结构，IPC 提升约 2.9%，相应的带宽开销却减少了约 61.2%。EIP128 相比 CTIP 的 IPC 性能提升约 2.5%，但相应带宽开销却增长约 282.8%。由此可知，CTIP 同时也是带宽友好型指令预取结构。

4.5 签名预测准确率

CTIP 结构利用签名触发相应的预取操作，所以签名的准确率也会直接影响预取的效率。其中，预测准确率 PA 的计算式如(1)所示。

$$PA=\frac{perfect_p refetch-prefetch_m iss}{perfect_p refetch+useless_p refetch},\qquad(1)$$

其中 *perfect_prefetch* 指的是完美预取所有函数调用缺失所需的最少预取总数；*prefetch_miss* 指的是由于预测失误而未能成功预取覆盖而造成缓存缺失的数量；*useless_prefetch* 指的是由于预测失误而错误预取造成无效缓存的数量。

各签名生成策略对应的预取准确率如表 9 所示，其中数据来源于 server_12, server_16, server_23, server_27, server_30 这 5 个具有代表性的、函数调用目标缺失率较大的服务器端程序。

Table 9 Comparison of Prefetch Accuracy of Each Signature Length

表 9 各签名长度预取准确率对比

签名长度	4	6	8	10
准确率/%	88.31	89.61	90.11	89.92
签名长度	16	32	{4, 8, 16, 32}	
准确率/%	89.49	89.49	91.86	

其中，在定长的签名生成策略中，签名长度为 8 时对应的预测准确率最高，约为 90.11%，而为签名长度为 4 对应的预测准确率最低，仅有 88.31%。而采用可变长的签名生成策略后，预测准确度达到了最高的 91.86%，这是任何定长签名策略都无法达到的预测准确率，比准确率最高的定长签名策略还高约 2 个百分点。对于长度较短的签名来说，它们的辨别能力较差，无法从已知条件中得出具体哪道预取才是要正确发出的，从而错误发出许多本不需要的预取请求，从而降低了预取准确率，造成带宽浪费以及缓存污染。而对于长度较长的签名来说，它们可能引入了不必要的噪声，区分度过高，拉长了学习周期，从而导致未能正确识别部分预取指令，造成了预取失效，削弱预取性能的同时可能浪费存储空间，因为噪声可能会占用无用的表项。而对于变长的签名策略来说，它们就很好地融合了两者的优点。因为其初始长度较短，所以不会由于引入噪声而增加学习

周期。又由于其在检测签名冲突后会向上分配长度更长的签名，从而不会由于签名长度过短识别不出来而导致预取冲突，冗余预取的问题。所以，其预取失效和无效预取请求的数量都控制在比较低的范围，使其最终的预测准确率高于任何一种定长签名的预取策略的预测准确率，准确率比定长签名策略高出2.6%左右。

4.6 其余策略及参数调整

4.6.1 位向量

CTIP结构中各位向量长度策略对应的IPC和预取请求数的关系如图12所示，其中IPC以及预取请求数都是通过运行server_11~server_13、server_16、server_23~server_25、server_27、server_31这几个比较具有代表性的服务器端程序得到的几何均值。

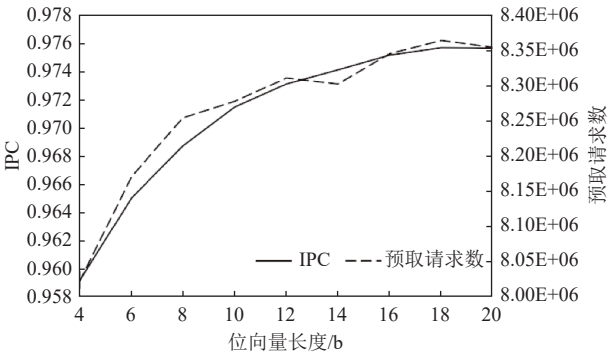


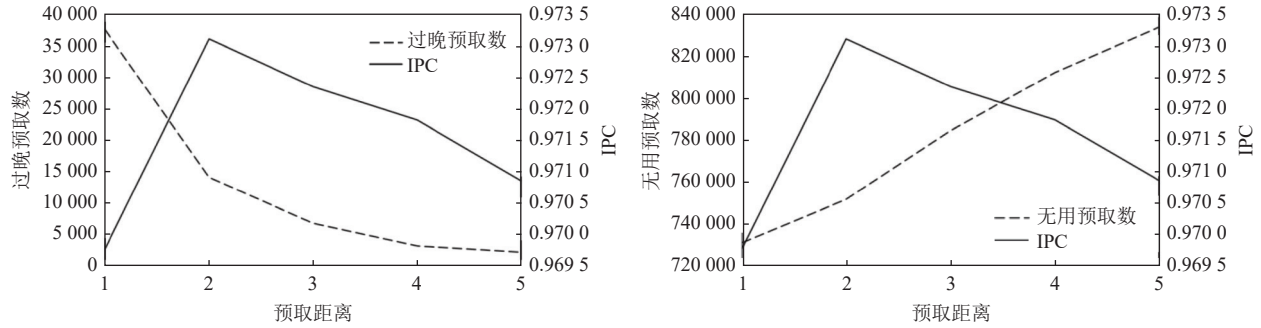
Fig. 12 Comparative diagram between IPC and the number of prefetch requests at the length of each bit-vector

图12 不同位向量长度下的IPC和预取请求数之间的对比图

由图12可见，IPC与预取请求数随着位向量长度的增长整体呈缓慢增长的趋势。在8 b前的预取请求数增长较快，在12 b前IPC增长较快。同时通过计算可知，每增加2 b位向量会给整体存储开销带来约0.3 KB的增长。过短的位向量可能导致部分空间区域没有被覆盖导致预取性能下降，而过长的位向量会导致额外的存储和带宽开销，甚至可能带来无效预取以及缓存污染等问题。结合IPC性能、预取请求数(带宽开销)以及存储开销间的权衡，CTIP结构选取位向量长度为12 b以达到综合较为理想的效果。

4.6.2 预取距离

CTIP结构中，各预取距离下IPC与过晚预取数的关系如图13(a)所示，IPC与无用预取数关系如图13(b)所示，其中IPC、过晚预取数以及无用预取数都是通过运行server_11~server_13、server_16、server_23~server_25、server_27、server_31这几个服务器端程序得到的几何平均值。由图13可知，当预取距离取2时能取得最优的IPC性能表现，且随着预取距离逐渐增大，过晚预取数量逐渐减少，减少幅度从距离1~2时最为明显；随着预取距离增大，无用预取数量逐渐增多，呈现相对比较平稳的趋势。过晚预取数随着预取距离增长逐渐减少的原因是，预取距离提高了导致相应的预取就会在更早的时刻发射，也更有可能掩盖相应的访存延迟而减少过晚预取数。如图1所示。无效预取数量随着预取距离增长逐渐增多，原因是当预取距离增大后，相应的缓存行就会更早到达指令缓存中，然而这会造成一个问题：该缓存行在第1次被程序访问前有可能就被其他缓存行提前置换出去，因为它提前太多被放入缓存中且一直没被访问过。因此这次预取没有带来任何实际收益，甚至带来了带宽浪费以及缓存污染，即造成无效预取。由此可见，过早或过晚发出预取都有可能带来性能衰退，通过选择合适的预取距离可以较好地避免以上2种情况的发生。结合图13的数据表现可知，在CTIP结构中取预取距离为2时可以得到最佳的IPC性能表现。



(a) 过晚预取数与IPC关系 (b) 无用预取数与IPC关系

Fig. 13 Relationship diagram between IPC, late prefetch and useless prefetch at different prefetch distances

图13 不同预取距离下IPC与过晚预取以及与无用预取的关系图

5 总 结

当今服务器端指令规模的不断扩大为一级指令缓存的存储空间持续带来巨大的挑战。指令预取技术在近些年来蓬勃发展，为处理器性能提升带来了很大贡献。但纵使有各式各样的预取结构被提出，指令预取仍有较大的发展前景和提升空间。其中一个有待解决的问题便是关于函数调用目标缺失率的优化。本文针对该研究方向，设计了特定的预取结构以及触发机制，利用较小的存储开销以及带宽消耗针对性地解决了服务器端函数调用目标缺失率较大的问题，同时 IPC 也得到了有效提升。

应用 CTIP 预取架构的缺失率比基准结构函数调用目标的缺失率低约 1.7%，比 EIP[12] 预取架构低约 45%，比 FDIP[38] 预取结构低约 95%；应用 CTIP 预取结构可达到最优的 IPC 性能表现，比基准结构的 IPC 提升约 11.9%，相同存储开销下比目前最先进的 EIP 预取器的 IPC 提升约 2.9%，同时降低带宽开销约 61.2%。该预取架构在服务器端程序有较好的应用前景，且呈现出与其他预取架构较好的正交性与可叠加性，有望在未来得到进一步发展。同时，该结构创新性引入了签名长度可动态化调整机制以及以位向量触发函数空间缓存块的预取机制，实现预取准确性以及存储效率的进一步提升。

在未来，还可以尝试将该预取结构与现有的 BTB 结构进一步结合，以寻求进一步提高存储效率，降低存储开销的同时提高 BTB 覆盖率；此外还可尝试引入预取距离可动态调节机制[12]，以响应预取及时性，避免过早或过晚预取，实现 IPC 性能表现的进一步提升。

作者贡献声明：陈立撰写论文，完成实验以及提出部分思路；高军提出实验思想和实验方案；赵天磊提出优化思路以及修改论文；刘峤提出指导意见并修改论文格式。

参 考 文 献

[1] Kumar R, Grot B, Nagarajan V. Blasting through the front-end bottleneck with shotgun[J]. ACM SIGPLAN Notices, 2018, 53(2): 30−42

[2] Gober N, Chacon G, Wang Lei, et al. The championship simulator: Architectural simulation for education and competition[J]. arXiv preprint, arXiv: 2210.14324, 2022

[3] Falsafi B, Wenisch T F. A Primer on Hardware Prefetching[M]. Berlin: Springer, 2022

[4] Reinman G, Calder B, Austin T. Fetch directed instruction prefetching[C]//Proc of the 32nd Annual ACM/IEEE Int Sym on Microarchitecture (MICRO-32). Piscataway, NJ: IEEE, 1999: 16−27

[5] Ferdman M, Wenisch T F, Ailamaki A, et al. Temporal instruction fetch streaming[C]//Proc of the 41st IEEE/ACM Int Symp on Microarchitecture. Piscataway, NJ: IEEE, 2008: 1−10

[6] Ferdman M, Kaynak C, Falsafi B. Proactive instruction fetch[C]// Proc of the 44th Annual IEEE/ACM Int Symp on Microarchitecture. Piscataway, NJ: IEEE, 2011: 152−162

[7] Kaynak C, Grot B, Falsafi B. Shift: Shared history instruction fetch for lean-core server processors[C]//Proc of the 46th Annual IEEE/ACM Int Symp on Microarchitecture. Piscataway, NJ: IEEE, 2013: 272−283

[8] Kaynak C, Grot B, Falsafi B. Confluence: Unified instruction supply for scale-out servers[C]//Proc of the 48th Int Symp on Microarchitecture. New York: ACM, 2015: 166−177

[9] Kumar R, Huang Cheng-chieh, Grot B, et al. Boomerang: A metadata-free architecture for control flow delivery[C]//Proc of the 2017 IEEE Int Symp on High Performance Computer Architecture (HPCA). Piscataway, NJ: IEEE, 2017: 493−504

[10] Ansari A, Lotfi-Kamran P, Sarbazi-Azad H. Divide and Conquer frontend bottleneck[C]//Proc of the 47th ACM/IEEE Annual Int Symp on Computer Architecture (ISCA). Piscataway, NJ: IEEE, 2020: 65−78

[11] Chacon G, Garza E, Jimborean A, et al. Composite instruction prefetching[C]//Proc of the 40th IEEE Int Conf on Computer Design (ICCD). Piscataway, NJ: IEEE, 2022: 471−478

[12] Ros A, Jimborean A. The entangling instruction prefetcher[EB/OL]. The First Instruction Prefetching Championship, 2020[2024-03-20]. https://research.ece.ncsu.edu/ipc/

[13] Jimenez D, Gratz P, Chacon G, et al. Barca: Branch agnostic region searching algorithm[EB/OL]. The First Instruction Prefetching Championship, 2020[2024-03-20]. https://research.ece.ncsu.edu/ipc/

[14] Nakamura T, Koizumi T, Degawa Y, et al. D-JOLT: Distant jolt prefetcher[EB/OL]. The First Instruction Prefetching Championship, 2020[2024-03-21]. https://research.ece.ncsu.edu/ipc/

[15] Seznec A. The FNL+ mma instruction cache prefetcher[EB/OL]. The First Instruction Prefetching Championship, 2020[2024-03-21]. https://research.ece.ncsu.edu/ipc/

[16] Gupta V, Kalani N S, Panda B. Runjump-run: Bouquet of instruction pointer jumpers for high performance instruction prefetching[EB/OL]. The First Instruction Prefetching Championship, 2020[2024-03-22]. https://research.ece.ncsu.edu/ipc/

[17] Ansari A, Golshan F, Lotfi-Kamran P, et al. Mana: Microarchitecting an instruction prefetcher[EB/OL]. The First Instruction Prefetching Championship, 2020[2024-03-22]. https://research.ece.ncsu.edu/ipc/

[18] Michaud P. Pips: Prefetching instructions with probabilistic scouts [EB/OL]. The First Instruction Prefetching Championship, 2020 [2024-03-23]. https://research.ece.ncsu.edu/ipc/

[19] Gober N, Chacon G, Jiménez D A, et al. Temporal ancestry prefetcher[EB/OL]. The First Instruction Prefetching Championship,

2020[2024-03-23]. https://research.ece.ncsu.edu/ipc/

[20] Gutierrez A, Dreslinski R G, Wenisch T F, et al. Full-system analysis and characterization of interactive smartphone applications[C]//Proc of the 2011 IEEE Int Symp on Workload Characterization (IISWC). Piscataway, NJ: IEEE, 2011: 81−90

[21] Seznec A, Michaud P. A case for (partially) tagged geometric history length branch prediction[J]. The Journal of Instruction-Level Parallelism, 2006, 8: 1−23

[22] Ferdman M, Adileh A, Kocberber O, et al. Clearing the clouds: A study of emerging scale-out workloads on modern hardware[J]. ACM SIGPLAN Notices, 2012, 47(4): 37−48

[23] VeidenbaumA V. Instruction cache prefetching using multilevel branch prediction[C]//Proc of the Int Symp on High Performance Computing. Berlin: Springer, 1997: 51−70

[24] Reinman G, Calder B, Austin T. Optimizations enabled by a decoupled front-end architecture[J]. IEEE Transactions on Computers, 2001, 50(4): 338−355

[25] Kolli A, Saidi A, Wenisch T F. RDIP: Return-address-stack directed instruction prefetching[C]//Proc of the 46th Annual IEEE/ACM Int Symp on Microarchitecture. Piscataway, NJ: IEEE, 2013: 260−271

[26] Ros A, Jimborean A. A cost-effective entangling prefetcher for instructions[C]//Proc of the 48th ACM/IEEE Annual Int Symp on Computer Architecture (ISCA). Piscataway, NJ: IEEE, 2021: 99−111

[27] Ros A, Jimborean A. Wrong-path-aware entangling instruction prefetcher[J]. IEEE Transactions on Computers, 2023, 73(2): 548−559

[28] Hassan M, Park C H, Black-Schaffer D. Protean: Resource-efficient instruction prefetching[C]//Proc of the 2023 Int Symp on Memory Systems. New York: ACM, 2023: 1−13

[29] Oh S, Xu Mingsheng, Khan T A, et al. Udp: Utility-driven fetch directed instruction prefetching[C]//Proc of the 51st ACM/IEEE Annual Int Symp on Computer Architecture (ISCA). Piscataway, NJ: IEEE, 2024: 1188−1201

[30] Godala B R, Ramesh S P, Pokam G A, et al. Pdip: Priority directed instruction prefetching[C]//Proc of the 29th ACM Int Conf on Architectural Support for Programming Languages and Operating Systems. New York: ACM, 2024: 846−861

[31] Lin Wenhai, Lin Yiquan, Chen Yiquan, et al. BTIP: Branch triggered instruction prefetcher ensuring timeliness[J]. Electronics, 2024, 13(21): 4323

[32] Khan T A, Brown N, Sriraman A, et al. Twig: Profile-guided BTB prefetching for data center applications[C]//Proc of the 54th Annual ACM/IEEE Int Sym on Microarchitecture (MICRO-54). Piscataway, NJ: IEEE, 2021: 816−829

[33] Khan T A, Sriraman A, Devietti J, et al. I-spy: Context-driven conditional instruction prefetching with coalescing[C]//Proc of the 53rd Annual IEEE/ACM Int Symp on Microarchitecture (MICRO-53). Piscataway, NJ: IEEE, 2020: 146−159

[34] Jamilan S, Khan T A, Ayers G, et al. Apt-get: Profile-guided timely software prefetching[C]//Proc of the 17th European Conf on Computer Systems. New York: ACM, 2022: 747−764

[35] Spracklen L, Chou Yuan, Abraham S G. Effective instruction prefetching in chip multiprocessors for modern commercial applications[C]//Proc of the 11th Int Symp on High-Performance Computer Architecture. Piscataway, NJ: IEEE, 2005: 225−236

[36] Annavaram M, Patel J M, Davidson E S. Call graph prefetching for database applications[J]. ACM Transactions on Computer Systems, 2003, 21(4): 412−444

[37] Asheim T, Grot B, Kumar R. Btb-x: A storage-effective BTB organization[J]. IEEE Computer Architecture Letters, 2021, 20(2): 134−137

[38] Ishii Y, Lee J, Nathella K, et al. Re-establishing fetch-directed instruction prefetching: An industry perspective[C]//Proc of the 2021 IEEE Int Symp on Performance Analysis of Systems and Software (ISPASS). Piscataway, NJ: IEEE, 2021: 172−182

[39] Ishii Y, Lee J, Nathella K, et al. Rebasing instruction prefetching: An industry perspective[J]. IEEE Computer Architecture Letters, 2020, 19(2): 147−150



Chen Li, born in 2000. Master candidate. His main research interest includes computer architecture.

陈　立, 2000 年生。硕士研究生。主要研究方向为计算机体系结构。



Gao Jun, born in 1978. PhD, senior engineer. His main research interest includes microprocessor design.

高　军, 1978 年生。博士，正高级工程师。主要研究方向为微处理器设计。



Zhao Tianlei, born in 1982. PhD, senior engineer. His main research interest includes high-performance processor design.

赵天磊, 1982 年生。博士，正高级工程师。主要研究方向为高性能处理器设计。



Liu Qiao, born in 1974. PhD, professor. His main research interests include machine learning and data mining.

刘　峤, 1974 年生。博士，教授。主要研究方向为机器学习、数据挖掘。