面向服务器的高存储效率分支目标缓冲器组织结构

Truls Asheim, NTNU，挪威

Boris Grot，爱丁堡大学，英国

Rakesh Kumar, NTNU，挪威

arXiv:2301.03899v1 [cs.AR] 10 Jan 2023

摘要——**许多当代应用的指令足迹达到数兆字节，超出分支目标缓冲器（BTB）和指令缓存（一级指令缓存）的容量，导致前端频繁停顿，进而不可避免地损害性能。BTB 容量对性能至关重要，因为足够大的 BTB 能让前端准确解析即将到来的执行路径，并恰当地引导指令取指。此外，它还能实现高效的取指引导型指令预取，从而消除很大一部分一级指令缓存缺失。因此，商用处理器为分支目标缓冲器分配了大量存储空间。**

本研究旨在通过优化 BTB 条目的组织结构，降低分支目标缓冲器的存储需求。我们的核心见解是：存储分支目标偏移量，而非完整目标或压缩目标，可以大幅降低分支目标缓冲器的存储开销，因为绝大多数动态分支的偏移量都很短，仅需少量位即可编码。基于这一见解，我们对组相联分支目标缓冲区的各路进行配置，使其存储不同位数的目标偏移量，从而让每一路存储特定范围内的偏移量。这样便能大幅减少目标地址所需的存储空间。我们的最终设计名为 BTB-X，采用八路组相联分支目标缓冲区，各路容量不同，因此与传统 BTB 相比，可跟踪的分支数量约多 2.24 倍；与存储优化的最先进分支目标缓冲器组织结构 PDede 相比，在存储预算相同的情况下，可跟踪的分支数量多 1.3 倍。

I. 引言

当代服务器应用程序的指令足迹庞大，源于层次深、结构复杂的软件栈。这些足迹可能远超分支目标缓冲器（BTB）和指令缓存（L1-I）的容量，从而造成所谓的前端瓶颈。BTB 未命中可能导致错误路径执行，并在检测到错误推测时触发流水线清空。这类流水线清空不仅会丢弃数十个周期的工作成果，还会暴露流水线的填充延迟。类似地，L1-I 缺失会使处理器核心前端停顿数十个周期，等待低级缓存处理缺失请求。

分支目标缓冲器之所以在高性能核心前端中居于核心地位，主要有三个原因：它决定要取指的指令流，为分支方向预测器识别分支，并影响一级指令缓存的命中率。具体而言，分支目标缓冲器通过识别控制流分歧，确保分支方向预测器能够对即将到来的条件分支进行预测。对于预测为跳转的分支和无条件分支，分支目标缓冲器会提供目标地址，以便将指令取指重定向至该地址。最后，分支目标缓冲器与方向预测器共同支持一类重要的指令预取器，即取指引导型指令预取器（FDIP） [30]–[32], [43], ，它们依靠分支目标缓冲器来发现一级指令缓存的预取候选项。

考虑到捕获现代工作负载的大型分支工作集至关重要，商用 CPU 配备了容量庞大的分支目标缓冲器， [24]也观察到了这一趋势。每个 BTB条目可能需要 8 字节或更多存储空间（第二节），因此分支目标缓冲器存储开销很容易达到数十甚至数百 KB。事实上，三星 Exynos M6 移动处理器为分支目标缓冲器分配了惊人的 529KB 片上存储空间 [21]。分支目标缓冲器存储开销不仅很高，而且还在快速增长。例如，在约八年间，三星 Exynos 的分支目标缓冲器存储预算从 M2 到 M6 增长了近六倍（从 98.9KB 增至 561.5KB） [21]。如此庞大的分支目标缓冲器虽然能有效捕获分支工作集，但代价是占用惊人的芯片面积。

为降低分支目标缓冲器存储空间的开销，先前的工作 [47]–[49] 着重压缩分支目标，因为如图 1 所示，分支目标占据了大部分 BTB 存储预算。具体而言，Seznec [47], [48] 指出，同一页内的所有分支目标共享相同的页码；与每个目标都存储一次页码相比，每页仅存储一次页码即可显著降低 BTB 的存储需求。为利用这一特点，他将 BTB 划分为两个结构：主分支目标缓冲器和页分支目标缓冲器，分别存储分支目标的不同部分。主分支目标缓冲器存储页偏移量，以及指向存储页码的页分支目标缓冲器条目的指针。最先进的分支目标缓冲器组织结构 PDede [49],进一步指出，目标地址所跨越的区域远少于页，其中区域是一组连续的页。因此，它进一步划分 BTB，并引入区域分支目标缓冲区，以降低页码的存储开销：区域内所有页的区域编号只需存储一次。通过为同一页或区域内的所有分支仅存储一次页码或区域编号，这些分支目标缓冲器避免了信息重复，从而降低了存储需求。

不过，尽管这些设计显著减少了分支目标缓冲器存储空间需求，却引入了若干复杂性，导致访问延迟和功耗需求增加。首先，这些设计引入了一层间接寻址，即每次访问分支目标缓冲器时，都要先访问主分支目标缓冲器，以获取指向页分支目标缓冲器和区域分支目标缓冲器的指针，然后才能访问这两个缓冲器。这种先后进行的访问（先访问主分支目标缓冲器，再访问页/区域分支目标缓冲器）增加了分支目标缓冲器的整体访问延迟，因此需要采用两周期的分支目标缓冲区查找，或延长时钟周期。这两种方案都会带来性能损失。其次，分配新的 BTB条目时，需要搜索页分支目标缓冲器和区域分支目标缓冲器，以检查目标地址对应的页/区域编号是否已存在。由于该页/区域编号可能位于页/区域分支目标缓冲区的任意位置，

有效位：1	标签：12	类型：2	目标：46	Rep_策略：3

图 1：传统 BTB 中一个条目的组成。数字表示编码各字段所需的位数。

因此需要进行全相联搜索 [48] ，这会增加分支目标缓冲器的功耗需求。另一种方案是限制页/区域编号在页/区域分支目标缓冲区中的存储位置 [49]；不过，这会增加发生冲突未命中的可能性。

本研究旨在降低分支目标缓冲器的存储需求，同时不增加其复杂度。为此，我们提出存储目标偏移量（定义为分支指令地址与其目标地址之间的差值），而非完整目标地址或压缩目标地址（即页偏移、页码和区域编号）。我们的核心洞见是，目标偏移量的分布并不均匀，但表示它们所需的位数通常远少于完整甚至压缩后的目标地址。分析表明，54% 的动态分支只需 6 位或更少即可编码其偏移量，而仅有 1% 的分支需要 25 位或更多来存储其偏移量。

基于这一洞见，我们提出在分支目标缓冲器中存储目标偏移量，而不是压缩或完整目标地址；后者的长度可能高达 64 位，具体取决于虚拟地址空间的大小。为适应目标偏移量分布的差异，我们设计组相联分支目标缓冲器的不同路，使其能够容纳不同数量的偏移量位；因此，每一路只存储目标偏移量可用特定位数编码的分支。这样做不仅能显著降低分支目标缓冲器的存储需求，还能避免当前最先进的 BTB 设计所具有的复杂性、间接寻址和全相联搜索。

本文介绍了 BTB-X，这是一种简单且极具存储效率的 BTB组织结构，体现了我们存储目标偏移量的理念。BTB-X 是一种组相联分支目标缓冲区，其各路的大小各不相同，用于存储不同位数的目标偏移量。我们的评估表明，在存储预算相同的情况下，与存储完整目标地址的传统 BTB 相比，BTB-X 能跟踪的分支数量约多 2.24 倍；与最先进的分支目标缓冲器组织结构 PDede 相比，则约多 1.3 倍。反过来，要容纳与传统 BTB 和 PDede 相同数量的分支，BTB-X 所需的存储空间分别少 2.24 倍和 1.3 倍。本工作的主要贡献如下：

- 我们指出，与存储完整或压缩后的目标地址相比，存储分支目标偏移量能够大幅节省分支目标缓冲器存储空间，因为约 54% 的分支只需 6 位或更少位数即可编码其偏移量。另有 22% 的分支需要 7 至 10 位。
- 我们发现，目标偏移量的位数分布并不均匀：分别需要 0–6 位、7–10 位和 11–25 位来编码 54%、22% 和 23% 的分支的偏移量。因此，采用统一位数的偏移量字段无法实现存储最优方案。
- 我们介绍 BTB-X，这是一种简单且存储效率极高的 BTB组织结构，它存储目标偏移量而非目标本身。此外，BTB-X 的各路容量经过设计，可存储不同大小的目标偏移量。
- 我们证明，在存储预算相同的情况下，BTB-X 可容纳的分支数量约为传统 BTB 的 2.24 倍、为最先进的 BTB——PDede 的 1.3 倍。我们的评估还表明，即使存储预算仅为传统 BTB 的一半，BTB-X 的表现仍优于传统 BTB。

II. 背景与动机

分支预测单元预测程序控制流，并向取指单元提供预测路径上的指令地址／程序计数器（PC）流；取指单元据此取出相应指令，供处理器核心的其余部分使用。由于分支指令会打断原本顺序执行的控制流，分支预测单元需要识别这些指令，以预测后续控制流。然而，一条指令是否为分支指令，只有在取指并解码后才能确定。为避免在生成下一个程序计数器之前取指并解码指令所带来的延迟，分支预测单元采用一种称为分支目标缓冲器（BTB）的专用硬件结构，仅凭程序计数器即可识别分支指令，甚至无需先取出指令本身。

A. 分支目标缓冲器（*BTB*）

图 1 展示了传统 BTB 的组织结构。每个 BTB 条目由有效位、标签、类型、目标和替换策略字段组成。图 1 还展示了这些字段通常所需的位数。标签 字段通常只存储部分标签，该部分标签通过对完整标签进行哈希处理生成，从而在尽量减少别名冲突的同时降低存储开销。目标 字段的位数取决于虚拟地址空间的大小和指令集架构（ISA）。我们假设虚拟地址空间为 48 位，指令集架构为 ARMv8，并据此计算图 1 中目标字段的大小。由于 ARMv8 指令始终为 32 位且按 4 字节对齐，程序计数器（PC）最低的两位始终为零。因此，目标字段只需 46 位。有效位 用于指示条目是否包含有效信息，而替换策略 位则用于在向 BTB 插入新分支时，选择一个已有分支进行驱逐。

要检查某个程序计数器是否对应一条分支指令，需使用程序计数器的低位对BTB进行索引，也就是访问BTB。程序计数器的高位则使用生成部分标签的同一函数进行哈希处理，并与索引到的BTB条目的标签字段进行比较。若匹配，则表示该程序计数器属于一条分支。

分支指令仅表示程序执行中可能存在控制流分歧点。然而，是否实际发生分歧取决于分支的类型，即调用、返回、条件分支或无条件分支，这些类型存储在 BTB 条目的类型 字段中。调用、返回和无条件分支总是会导致控制流分歧，因为它们总是被采用。

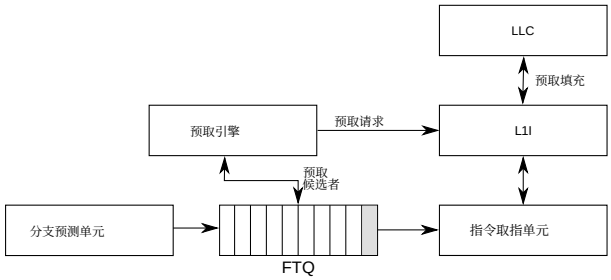


图 2：FDIP微架构

相比之下，条件分支并非总是被采用，因此会使用方向预测器来预测其方向。

如果预测某个分支将被采用，BTB条目中的目标字段会提供下一条指令的地址，但返回指令除外。这是因为返回地址取决于调用点，而同一个函数可以从不同的调用点调用。因此，通常会使用返回地址栈（RAS）来记录各调用点的返回地址。函数调用时，调用指令会将返回地址压入 RAS，随后由对应的返回指令弹出。

B. BTB 未命中的代价

分支指令发生 BTB 未命中，意味着未检测到该分支，前端会继续顺序取指。顺序路径是否正确，取决于未命中分支的实际方向。除非未命中的是一个未被采用的条件分支，否则顺序路径就是错误的。当处理器核心最终检测到走错路径时，BTB 未命中分支之后的所有指令都会被清除，取指会重定向到分支目标，并用正确路径上的指令填充流水线。因此，BTB 未命中会严重损害性能，因为它们会导致数十个周期的工作白白损失，并暴露流水线填充延迟。

C. 分支目标缓冲器在指令预取中的作用

取指引导型指令预取器是一类功能强大的一级指令缓存预取器，其本质上依赖分支目标缓冲器来识别预取候选项。这类预取器效果显著；当配备足够大的分支目标缓冲器时，其性能优于最近结束的指令预取竞赛的获胜方案 [1] ，并接近理想一级指令缓存的性能，正如 Ishii 等人所报告的那样 [24]。这些预取器的变体已应用于商业产品，例如 IBM z15 [45]、ARM Neoverse N1 [41]等。

图 2 展示了取指引导型指令预取器（FDIP） [43] 的典型组织结构。如最初提出的那样，FDIP 通过取指目标队列（*FTQ*）将分支预测单元与取指引擎解耦。这种解耦使分支预测单元能够领先于取指引擎运行，通过预测远期控制流来发现预取候选项。采用 FDIP 时，分支预测单元每个周期都会识别并预测分支，以预判即将到来的执行路径，并将相应的指令地址插入

表 I：三星 Exynos 处理器中的分支目标缓冲器存储开销

CPU	分支目标缓冲器存储空间
M1/M2	98.9KB
M3	175.8KB
M4	288.0KB
M5	310.8KB
M6	561.5KB

FTQ。因此，FTQ 中包含一串处理器核心将要取指的预期指令地址。预取引擎会扫描 FTQ，以识别预取候选项并发出预取请求。

要使 FDIP 发挥效用，分支目标缓冲器必须能够容纳分支工作集，否则频繁的 BTB 未命中会导致 FDIP 预取错误路径，因为 FTQ 中会填入错误路径的指令地址。这是商用处理器采用大容量分支目标缓冲器的关键原因之一，[24]也观察到了这一点。这些大容量分支目标缓冲器会带来惊人的存储开销。此外，分支目标缓冲器存储开销不仅很高，还在迅速增长。例如，表 I 列出了几代三星 Exynos处理器的分支目标缓冲器存储开销。如表所示，除 M4 到 M5 之间外，每一代的分支目标缓冲器存储开销几乎都翻了一番。总体而言，约八年间，存储开销从 M1 到 M6 增长了近六倍（从 98.9KB 增至 561.5KB） [21]。

随着服务器应用程序的指令占用空间不断扩大，Google网页搜索工作负载也呈现出这一趋势，其指令足迹正以年化 27% 的速度增长 [25], ，未来分支目标缓冲器的容量及其存储开销必将增加。因此，迫切需要研究高存储效率的分支目标缓冲区组织结构，以应对前端瓶颈，同时避免耗费过多的面积预算。

III. 分支目标距离分析

分支目标的存储开销占分支目标缓冲器存储需求的很大一部分。例如，在图 1 所示的传统 BTB 中，目标字段约占分支目标缓冲器总存储需求的 72%（64 位中的 46 位）。我们分析存储分支目标偏移量 所需的位数，以评估存储偏移量而非完整或压缩目标是否能够减少分支目标缓冲器存储需求。我们将目标偏移量定义为目标地址的n个最低有效位，其中n 是分支 PC 与目标之间出现差异的最高有效位的位置。例如，对于图 3 所示的分支 PC 和目标，它们之间出现差异的最高有效位位于第 5 位，而所有高于第 5 位的位都相同。

位位置	48	...	9	8	7	6	5	4	3	2	1
分支 PC	0	...	1	0	1	1	0	1	0	0	0
分支目标	0	...	1	0	1	1	1	1	0	0	0
目标偏移量		...					1	1	0	0	0

图3：分支目标偏移量示例

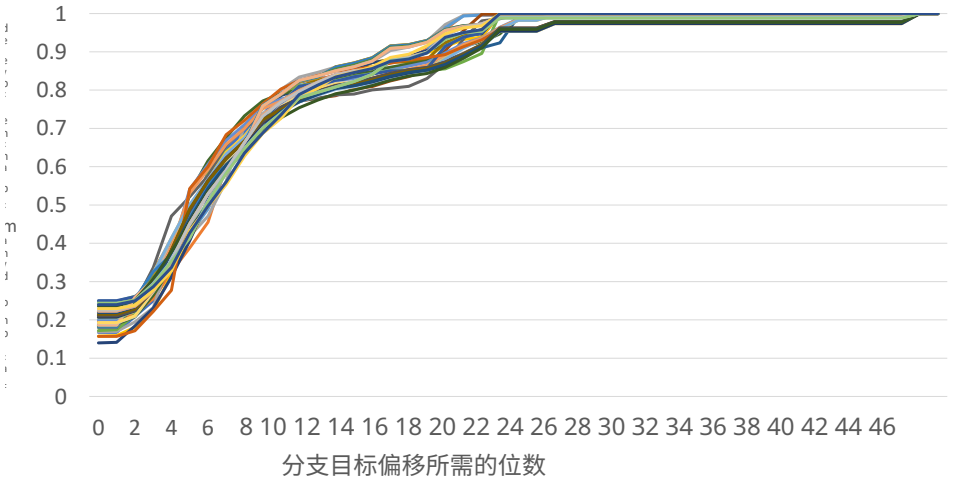


图 4：不同工作负载中分支目标偏移量的分布。

因此，这对分支 PC 和目标的目标偏移量为“11000”，即目标从第 5 位到第 1 位的各位。此外，由于我们建模的 ARM v8 指令集架构将指令对齐到 4 字节边界，目标地址的最低两位始终为零。因此，我们只需在分支目标缓冲器中将“110”作为偏移量存储。

将偏移量定义为n 位低位目标地址，而不是分支 PC 与目标地址之间的数值距离（即 target-PC），其优势在于，只需将移位后的分支 PC 与从分支目标缓冲器中检索出的偏移量拼接起来，就能恢复完整目标地址。相比之下，若将数值距离用作偏移量，则需要一个 48 位加法器才能从偏移量恢复完整目标地址。

图 4 展示了我们的工作负载中分支工作集的分支目标偏移量分布。数据涵盖条件分支和无条件分支，因此全面覆盖了整个分支工作集。X 轴表示存储偏移量所需的位数，Y 轴则表示所覆盖的动态分支比例。

如图所示，分布以短偏移量为主：54% 的分支所需的偏移量不超过 6 位，另有 22% 的分支仅需 7 至 10 位来表示其偏移量。偏移量如此之短的主要原因是，条件分支在动态分支工作集中占主导地位，而且其偏移量通常较短 [32]。这是因为条件分支通常只在函数内部引导控制流；与此同时，软件工程原则倾向于使用较小的函数，因此条件分支的目标偏移量距离较短。此外，如第二节所述，返回指令从返回地址栈获取其目标，因此无需在分支目标缓冲器中存储任何目标位。因此，在本分析中，我们假设返回指令的偏移量为 0 位。

或许令人惊讶的是，图 4 还显示，只有极少数分支的偏移量需要占用大量位。事实上，偏移量超过 25 位的分支仅占 1%。综合这些结果可知，预留空间来存储完整的 46 位目标地址，会导致分支目标缓冲器存储空间利用率极低，因为如果改用偏移量，99% 的分支最多只需要表示完整目标地址所需位数的一半。

通过这项分析，我们得出了两个关键见解：

关键见解 1：大多数分支的目标在虚拟地址空间中都离该分支本身相对较近。因此，将到目标的距离表示为相对于分支指令的偏移量，可以大幅节省存储空间。

关键洞察 2：目标偏移量的大小分布不均：分别需要 0–6 位、7–10 位和 11–25 位来编码 54%、22% 和 23% 的分支的偏移量。因此，采用单一大小的偏移量字段无法实现存储开销最优的解决方案。

IV. 最先进的分支目标缓冲器及其局限性

以往的研究提出了多种 BTB 组织结构，通过压缩分支目标来降低存储开销。本节介绍最具代表性的 BTB 组织结构，并分析其局限性。

A. 缩减型分支目标缓冲器：

Seznec [47]发现了一个关键现象：同一页内的所有分支目标共享相同的页码，区别仅在于页内偏移量。因此，在 BTB 中存储完整目标地址会导致页码大量重复，浪费存储容量。为消除这种重复，Seznec 提出了缩减型分支目标缓冲器（R-BTB），其变体也用于 ITTAGE [48]。R-BTB 的关键创新在于存储指向页码的指针，而不是在 BTB 中直接存储页码。

图5展示了 R-BTB 的逻辑组织及其条目的组成。R-BTB 由两个分区组成：主分支目标缓冲器和页分支目标缓冲器。对于每个分支目标，除页偏移外，主分支目标缓冲器还存储指向页码的指针。页码本身存储在页分支目标缓冲器中。如果两个或更多分支的目标位于同一页中，它们的

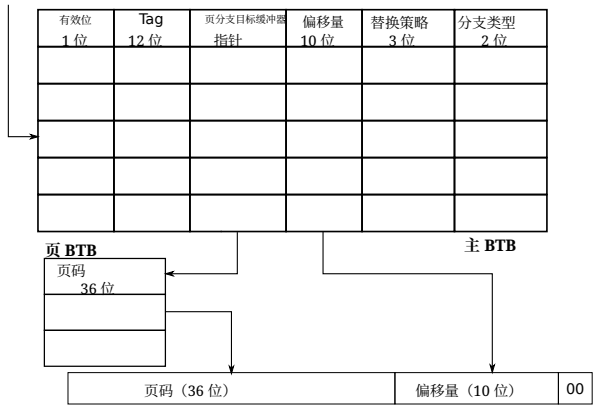


图 5：缩减型分支目标缓冲器组织结构。

主分支目标缓冲器条目将持有指向同一页分支目标缓冲器条目的指针。由于页的数量远少于分支目标的数量，存储页分支目标缓冲器指针所需的位数少于存储页码本身所需的位数。因此，通过在页分支目标缓冲器中仅存储一次页码，R-BTB 避免了重复并减少了存储需求。

B. PDede：

PDede [49]是最先进的分支目标缓冲器组织结构，包含三种不同的变体。图 6 展示了存储效率最高、性能最佳的 PDede 变体，称为 PDede 多条目大小。它在两个方面优于 R-BTB。首先，它降低了在页分支目标缓冲器中存储页码的成本。PDede 注意到，服务器应用程序由于指令占用空间较大，会访问大量页面，从而增加页分支目标缓冲器的存储需求。他们还发现，由于不同的库会被动态映射到地址空间中的不同位置，这些页面往往会形成空间上的区域，其中一个区域由多个连续页面组成。正如同一页面内的分支目标共享相同的页码，区域内的页码也共享相同的区域编号。为消除区域编号的重复，如图 6 所示，PDede 引入了区域分支目标缓冲区来存储区域编号，而主分支目标缓冲器则存储指向它的指针，就像它存储指向页 BTB 的指针一样。

其次，对于同页分支，即分支及其目标位于同一页时，PDede 不存储页/区域编号，因为这些编号可从分支 PC 中恢复。PDede 为组相联 BTB 中的同页分支预留一半的路。如图 7 所示，为同页分支预留的路无需存储页分支目标缓冲器和区域分支目标缓冲区指针，因此 PDede 进一步节省了存储空间。

C. 当前最先进技术的局限性：

尽管 R-BTB 和 PDede 通过避免重复存储页和区域编号实现了显著的存储节省，但它们引入了一级间接寻址以及在页分支目标缓冲器和区域分支目标缓冲区中的相联搜索，从而增加了分支目标缓冲器的复杂性。这些

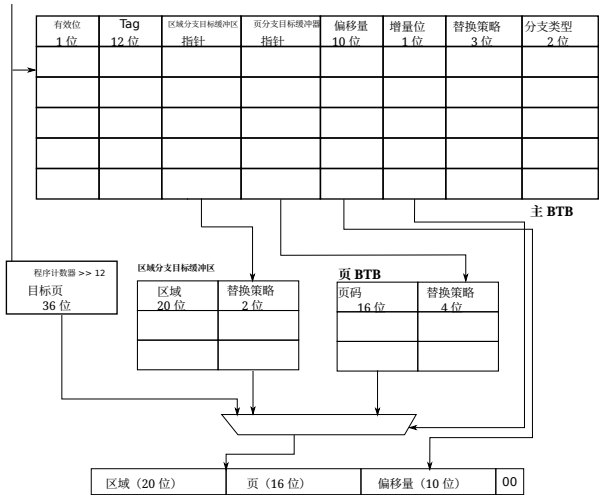


图 6：PDede BTB 组织结构。



图 7：异页与同页 PDede 条目的组成。

复杂性导致访问延迟增加，并带来更高的功耗需求。此外，当前最先进的 BTB 设计在利用可用存储预算方面并不理想。

间接寻址：如图 5 和图 6 所示，对主分支目标缓冲器的访问只能提供目标地址的一部分，即页偏移量。其余部分必须从页分支目标缓冲器和区域分支目标缓冲区中获取。此外，由于主分支目标缓冲器的访问会提供指向它们的指针，因此无法与主分支目标缓冲器并行访问页分支目标缓冲器和区域分支目标缓冲区。因此，依次访问主分支目标缓冲器和页/区域分支目标缓冲器会增加整体分支目标缓冲器访问延迟。这种额外延迟要么迫使分支目标缓冲器查找耗费两个周期，要么需要延长时钟周期。这两种方案都会损害性能。PDede 在一定程度上避免了这种间接寻址惩罚，因为同页分支无需访问页/区域分支目标缓冲器，而是直接从分支 PC 获取页编号和区域编号。不过，异页分支，即分支 PC 与目标地址位于不同页的分支，仍需承受间接寻址惩罚。

相联搜索：分配新的BTB条目时，可能需要更新所有BTB分区（主分支目标缓冲器、页分支目标缓冲器和区域分支目标缓冲区）。替换策略会选择主分支目标缓冲器中要替换的条目。然而，需要搜索页分支目标缓冲器和区域分支目标缓冲区，以检查传入目标的页码/区域编号是否已存在。由于页码/区域编号可能出现在页/区域分支目标缓冲区的任何位置，ITTAGE [48] 使用全相联搜索，这会增加功耗需求，尤其是在条目数量增加时。PDede通过限制给定分支目标的页码可驻留的条目数量（部分）解决了这一局限，将其限制为16个。不过，

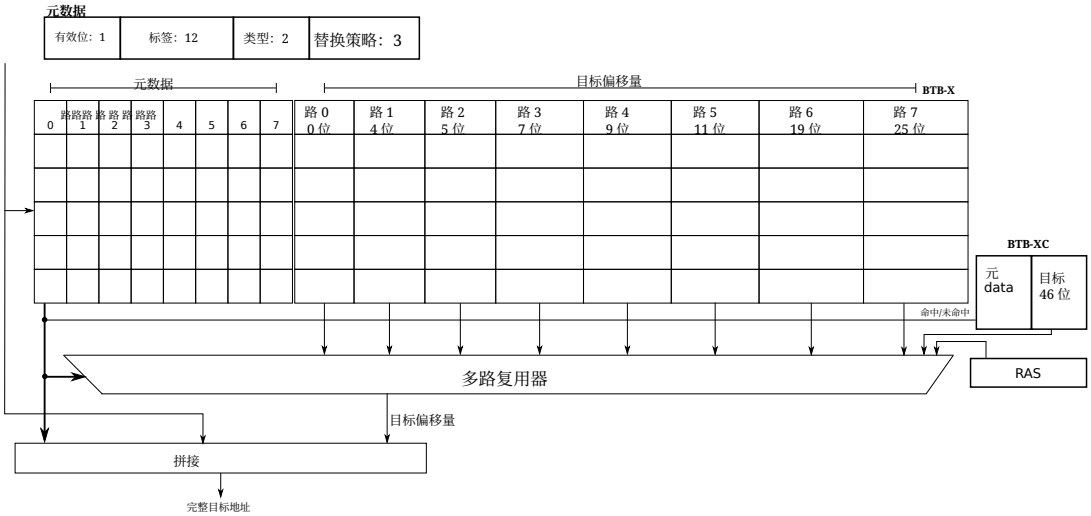


图 8：BTB-X 组织结构及条目/组构成。

限制条目数量会增加发生冲突未命中的可能性。

存储利用率欠佳：尽管 R-BTB 和 PDede 相较于传统 BTB 显著提高了存储利用率，但仍有很大的改进空间。这是因为，R-BTB 对所有分支都采用固定大小的目标表示，即 10 位偏移量加上固定大小的页 BTB 指针。相比之下，第二节中的分析表明，目标偏移量的大小存在很大差异，因此 R-BTB 采用单一大小的存储组织效率低下，因为其必须按照最大的目标偏移量来确定大小。例如，如图 4 所示，54% 的目标偏移量不超过 6 位；然而，R-BTB 对这些分支仍需使用全部 10+ 位，因而导致存储利用率低下。PDede 的存储利用率略高于 R-BTB，因为它为同页和跨页分支设置了不同大小的条目。然而，只有两种不同大小的 BTB 条目（即同页和跨页）不足以应对服务器应用程序中观察到的目标偏移量大小的巨大差异（图 4）。

V. BTB-X

BTB-X 是一种简单且存储效率高的 BTB 组织结构。基于第三节的研究发现，它存储目标偏移量而非完整目标地址，以尽量减少存储需求，同时兼顾目标偏移量大小的巨大差异。其微架构以及条目/组构成如图 8 所示。

A. BTB-X 组织结构

BTB-X 中的偏移量字段需要能够适应第三节中观察到的目标偏移量大小的不均匀分布。原则上，该字段的位数应足以存储最大的目标偏移量。然而，由于最大偏移量可能几乎与完整目标地址一样大，若按此方式设置偏移量字段的位数，存储偏移量所能节省的存储空间几乎就会荡然无存。一个颇具吸引力的替代方案是，将偏移量字段的位数设为能够存储大多数偏移量。观察图 4 可知，25 位的偏移量字段可以覆盖 99% 以上的分支目标偏移量，因为这些偏移量所需位数不超过 25 位。不过，这种方案有两个主要缺点。首先，它仍会导致存储利用率低下。这是因为，如图 4 所示，54% 的分支会浪费超过四分之三的偏移量存储容量，因为它们的偏移量只需 6 位或更少。另有 25% 的分支会浪费近一半的偏移量存储空间，因为它们的偏移量只需 7 到 12 位。其次，所有偏移量超过 25 位的分支都无法装入 BTB，因而总会导致 BTB 未命中。不过，这类分支非常少（¡ 1%），因此其影响可能很小。

为尽量减少存储空间利用不足，我们为组相联 BTB-X 中一个组的不同路配置不同大小的目标偏移量字段。分支会被分配到偏移量字段至少能容纳其目标偏移量所需位数的那一路。我们采用八路组相联 BTB-X，并利用图 4 中的数据合理设定每一路的偏移量字段大小，使每一路覆盖约 12.5% 的动态执行分支。图 4 显示，平均而言，0、4、5、7、9、11、19 和 25 位偏移量分别覆盖约 20%、36%、46%、61%、72%、79%、90% 和 99% 的动态分支。因此，我们将 BTB-X 的八路分别配置为可存储 0、4、5、7、9、11、19 和 25 位的目标偏移量。请注意，约 20% 的动态分支所需的偏移量为 0 位，这些分支是从返回地址栈读取目标地址的返回指令，如第二节所述。因此，BTB-X 的第 0 路不为目标偏移量提供任何存储空间。尽管返回指令不从 BTB 获取目标地址，但仍需为其分配 BTB 条目，以便分支预测单元能够识别这些指令，并在生成待取指的指令流时从返回地址栈中选取其目标地址。

BTB-X 覆盖了动态执行分支的 99%，我们采用一个非常小的传统直接映射分支目标缓冲器，称为 BTB-XC，用于存储

其余 1% 分支的完整目标地址。为这类分支在 BTB-X 中预留一路会不必要地增加存储需求，因为这类分支所需的条目远少于 BTB-X 的组数。事实上，根据我们的分析，我们将 BTB-XC 的容量设计为 BTB-X 的 1/64，即条目数是 BTB-X 组数的 1/8。

B. 访问 *BTB-X*

BTB-X 查找： BTB-X 查找与图 8 所示的传统 BTB 查找非常相似。它使用程序计数器的索引位进行访问，并行查找全部八路。此外，BTB-XC 也与 BTB-X 并行查找。传统 BTB 查找与 BTB-X 查找的主要区别在于，如果在第 1 路至第 7 路命中，BTB-X 查找提供的是目标偏移量，而非完整目标地址。因此，需要将目标偏移量与分支 PC 拼接，以获得完整目标地址。需要从分支 PC 拼接的位数取决于查找命中的 BTB-X 路。例如，命中第 1 路时会提供目标地址低位的 4 位，其余部分则需要从分支 PC 拼接。此外，命中第 0 路表示完整目标地址位于返回地址栈中，而命中 BTB-XC 则会提供完整目标地址。

BTB-X分配：与任何现有的BTB组织结构一样，分支指令退役时会分配（或更新）BTB-X条目。表示分支目标偏移量所需的位数决定了分支可以在哪些路中分配条目。例如，返回指令可以根据替换策略的决定，在任意一路中分配条目，因为它们没有偏移量，可以适用于所有路。其他分支可分配条目的路数较少，具体取决于存储其偏移量所需的最少位数。例如，如果某个分支的目标偏移量需要20位，则不能在第0路到第6路中为其分配条目。

BTB-X采用略作修改的最近最少使用（LRU）替换策略。具体而言，我们只比较能够容纳目标偏移量的条目的LRU计数器，并在其中替换最近最少使用的条目。LRU的其他所有方面，例如计数器更新，都与基准策略完全相同。

VI. 评估

A. 方法

我们使用ChampSim [4] ，根据高通为首届指令预取竞赛（IPC-1） [1]提供的服务器和客户端工作负载轨迹，评估BTB-X的效果。我们先对微体系结构状态预热5000万条指令，再统计接下来5000万条指令的数据。表II列出了所建模处理器的微体系结构参数，该处理器仿照英特尔Sunny Cove [5]。

我们改进了 Champsim 的两个重要方面，以评估基准、最先进和所提出的 BTB 组织方式。首先，作为一种基于跟踪的模拟器，Champsim 通过查询跟踪中提供的信息来检测分支，而不是查找 BTB。这实际上

表II：微体系结构参数

参数	值
取指	6 路、128 条指令的 FTQ
分支预测器	哈希感知器
返回地址栈	64 个条目
调度器	128 条目
重排序缓冲区	352 条目
加载队列	128 条目
存储队列	72 个条目
L1-I	32 KB，8 路， 4 周期延迟，8 个未命中状态处理寄存器
L1-D	48 KB，12 路， 5 周期延迟，16 个未命中状态处理寄存器
L2	512 KB，8 路， 14/15 周期延迟，32 个未命中状态处理寄存器
LLC	2 MB，16 路， 34/35 周期延迟，64 个未命中状态处理寄存器

表 III：BTB-X 存储需求。括号中的数字适用于 BTB-XC。

条目	Sets	组大小	存储
256(4)	32(4)	224(64)-位	0.9KB
512(8)	64(8)	224(64)-位	1.8KB
1K(16)	128(16)	224(64)位	3.6KB
2K(32)	256(32)	224(64)位	7.25KB
4K(64)	512(64)	224(64)-位	14.5KB
8K(128)	1024(128)	224(64)位	29KB
16K(256)	2048(256)	224(64)位	58KB

意味着 Champsim 使用的是理想 BTB。因此，我们首先在 Champsim 中实现了第二节介绍的现实传统 BTB（Conv-BTB）。其次，Champsim 会在执行阶段解析所有分支，也就是说，只有当发生预测错误的分支指令到达执行阶段时，才会检测到分支预测错误，并将取指重新引导至正确路径。这种分支解析方式高估了无条件直接分支的预测错误惩罚。这是因为这类分支始终会跳转，因此可以在译码阶段解析（从而更早地重新引导取指）：通过将下一条指令的程序计数器与分支指令中编码的目标进行比较，即可检测预测错误。此外，对于 BTB 未命中但方向预测器正确预测为跳转的已跳转条件分支，也可以在译码阶段解析。为此，无论 BTB 命中还是未命中，取指阶段都会将所有指令的方向预测传递到译码阶段。如果译码阶段发现某条分支指令在 BTB 中未命中，但方向预测器预测其会跳转，就会将取指重新引导至该分支指令中编码的目标，从而降低 BTB 未命中惩罚。鉴于方向预测器的准确率很高，这项优化能够降低平均 BTB 未命中惩罚。总体而言，我们改进了分支解析机制，使在 BTB 中未命中的无条件直接分支和已跳转条件分支都能在译码阶段解析。最后，BTB 仅在提交阶段使用已跳转分支（包括条件分支和无条件分支）进行更新。

B. 存储分解

我们首先评估在相同存储预算下，不同 BTB 组织（Conv-BTB、PDede 和 BTB-X）可容纳的分支数量

表 IV：不同存储预算下各 BTB 组织方式中的分支数量 s.

存储	BTB-X + BTB-XC	PDede				Conv-BTB	
	分支	页级 BTB 预算	主 BTB 预算	条目大小	分支	条目大小	分支
0.9KB	256 + 4	0.078KB	0.817KB	32 位	210	64 位	116
1.8KB	512 + 8	0.156KB	1.645KB	32.5 位	415	64 位	232
3.6KB	1K + 16	0.312KB	3.3KB	33 位	820	64 位	464
7.25KB	2K + 32	0.625KB	6.6KB	33.5 位	1617	64 位	928
14.5KB	4K + 64	1.25KB	13.2KB	34 位	3190	64 位	1856
29KB	8K + 128	2.5KB	26.5KB	34.5 位	6292	64 位	3712
58KB	16K + 256	5KB	53KB	35 位	12405	64 位	7424

并相互比较。我们采用表 III 中所示的 BTB-X 存储预算，该预算分别用于存储 256、512、1K、2K、4K、8K 和 16K 个分支。计算假设虚拟地址空间为 48 位，且 BTB-X 条目组成如图 8 所示。若要将 BTB-X 中的条目数翻倍，我们会将组数翻倍，同时保持相联度不变。请注意，表 III 列出的是组大小，而非条目大小。这是因为 BTB-X 在不同路中采用了大小不同的条目；不过，组大小保持不变。

表 IV 展示了不同 BTB 组织结构在不同存储预算下可跟踪的分支数量。PDede 将总的分支目标缓冲器存储空间预算分配给其主分支目标缓冲器、页分支目标缓冲器和区域分支目标缓冲区。我们按照其发明者 [49]采用的分配方式，在不同 PDede 分支目标缓冲器之间分配预算，如表 IV 所示。因此，对于 29KB 的存储预算，我们将 PDede 配置为使用 1K 个页分支目标缓冲器条目和约 6K 个主分支目标缓冲器条目。将存储预算减半至更低水平时，我们也将主分支目标缓冲器和页分支目标缓冲器中的条目数减半。页分支目标缓冲器条目数减半后，主分支目标缓冲器中存储页 BTB 指针所需的位数也会减少。因此，随着存储预算降低，主分支目标缓冲器条目大小也会减小。此外，在所有存储预算下，我们都使用四个区域分支目标缓冲区条目，因此区域分支目标缓冲区需要固定的 0.0107KB 存储空间。还需注意，PDede 为同页分支预留了每组一半的路，而另一半既可存储同页分支，也可存储异页分支。因此，其条目有两种不同的大小。表 IV 中所示的 PDede 条目大小是这两种大小的平均值。

如表所示，BTB-X 存储的分支数量显著多于其他任何 BTB 组织方式。具体而言，其存储的分支数量是传统 BTB 组织结构的 2.24 倍。与 PDede 相比，在 0.9KB 存储预算下，BTB-X 存储的分支数量多 1.24 倍；在 58KB 存储预算下，则多 1.34 倍。随着存储预算增加，BTB-X 相较于 PDede 的优势也随之扩大，因为在预算较高时，PDede 条目需要更多位来容纳更大的页 BTB 指针。

C. BTB 每千条指令未命中数

为了了解更高 BTB-X 分支密度的优势，我们测量了不同 BTB 组织结构在客户端和服务器工作负载上产生的每千条指令未命中数（MPKI）。由于未跳转分支的 BTB 未命中不会影响性能，因此我们只考虑已跳转分支的 BTB 未命中。在此分析中，我们假设 BTB 分支目标缓冲器存储空间预算为 14.5KB，对应于 4160、3190 和 1856 个条目分别对应 BTB-X、PDede 和 Conv-BTB。结果如图 9 所示。

如图所示，由于服务器工作负载的指令和分支足迹都非常庞大，其 MPKI 显著高于客户端工作负载。图中还显示，与传统 BTB 和 PDede 相比，BTB-X 的 MPKI 要低得多，尤其是在服务器工作负载上。具体而言，对于服务器工作负载，传统 BTB 平均产生 25 MPKI，因为在给定存储预算下，它存储的分支数量是三种组织方式中最少的。PDede 可将 MPKI 降至 13.7，而 BTB-X 则进一步降至 9.5。BTB-X 相对于其他组织方式的优势在 MPKI 特别高的工作负载上尤为明显，例如服务器023至服务器035；在这些工作负载上，BTB-X 的 MPKI 远低于传统 BTB 和 PDede。

D. 性能

为评估 MPKI 降低如何转化为性能提升，我们比较了三种 BTB 组织方式在客户端和服务器工作负载上的性能。回顾第二节，更大的 BTB 带来两项截然不同的优势：1）通过检测即将到来的控制流中的分支，减少流水线刷新的发生；2）与 FDIP 配合时，促进指令预取。因此，我们通过结合 FDIP 对三种 BTB 组织方式进行评估，以比较它们带来的性能收益。

图10展示了在服务器和客户端轨迹上获得的性能收益。结果以未进行指令预取的 Conv-BTB 性能为基准进行归一化。图中每个工作负载对应三根柱状条。第一根柱状条表示 Conv-BTB 与 FDIP 配合时获得的性能收益。第二和第三根柱状条分别表示 PDede 和 BTB-X 获得的性能收益。PDede 和 BTB-X 的柱状条将性能收益分解为两部分：减少流水线刷新所带来的收益，以及由于 BTB 收录了更多分支而改善指令预取所带来的收益。

从搭配指令预取器（FDIP）时的整体性能收益来看，图中显示，在服务器工作负载上，BTB-X 相较于基准的几何平均性能收益为 39%。相比之下，PDede 和 Conv-BTB 在这些工作负载上的性能收益仅为 33% 和 24%。从各个工作负载来看，在服务器023至服务器032上，BTB-X 全面优于 PDede 和 Conv-BTB。例如，在服务器032上，BTB-X 相较于基准实现了 83% 的加速比，而 PDede 和 Conv-BTB 的性能收益仅为 60% 和 32%。这是因为分支

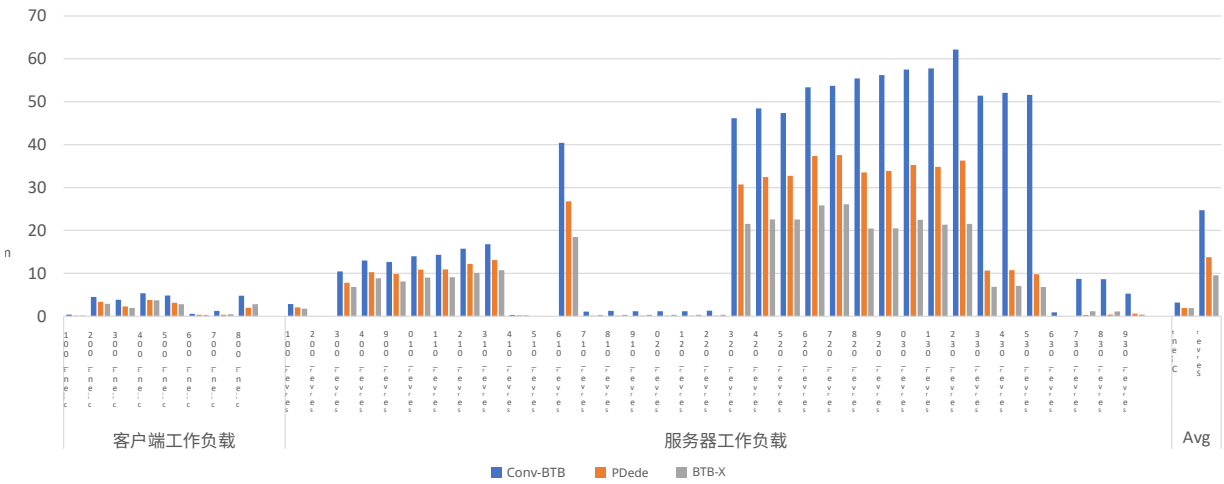


图 9：不同 BTB 组织方式下的 BTB 每千条指令未命中数。

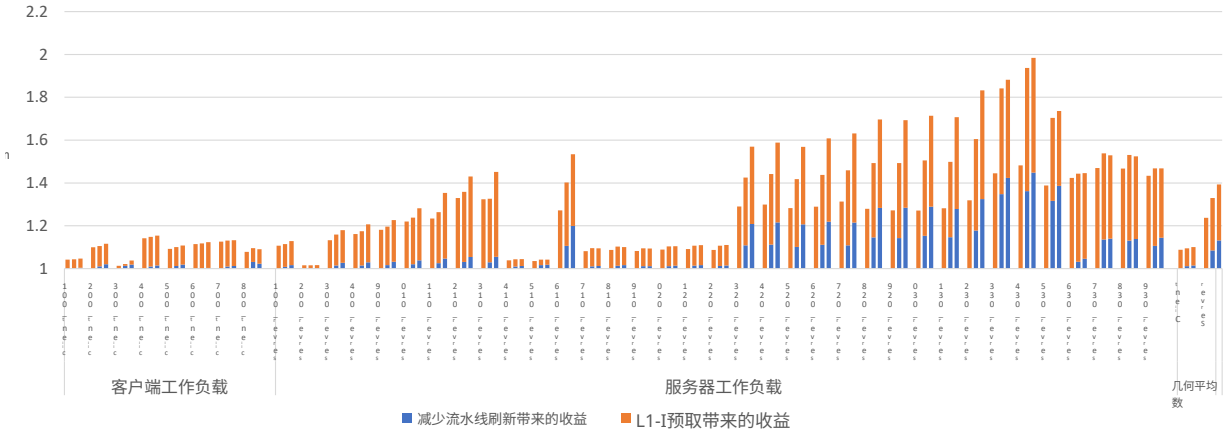


图 10：与不使用 FDIP 的传统 BTB 相比，传统 BTB（使用 FDIP）、PDede 和 BTB-X（使用和不使用 FDIP）所获得的性能收益。每个工作负载对应的三个柱分别为 Conv-BTB、PDede 和 BTB-X。

由于 BTB-X 的分支容量更大，这些工作负载的分支工作集开始能够装入 BTB-X。因此，BTB 每千条指令未命中数降低，这不仅减少了流水线清空，还使 FDIP 能够在更长时间内保持在正确的预取路径上。

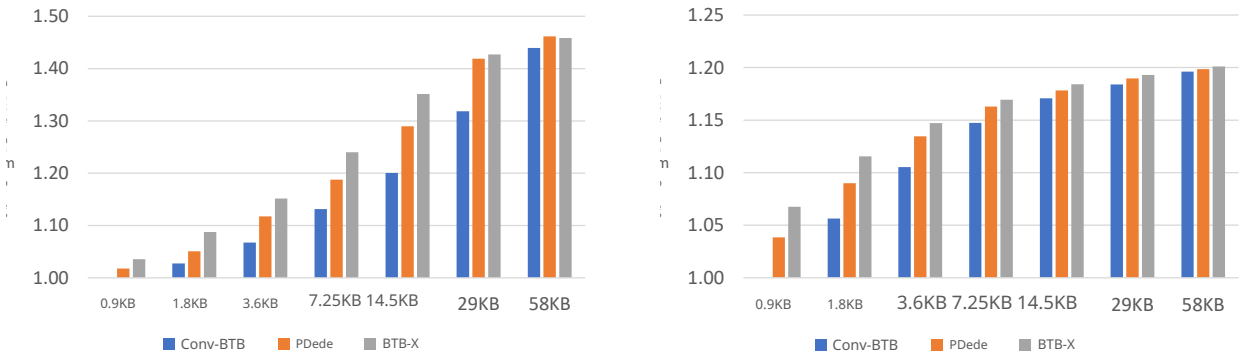
观察不使用指令预取器时的结果，图 10 显示，BTB-X 相较于基线 Conv-BTB 的性能收益为 13%，而 PDede 的收益为 8%。在单个工作负载上，即使不使用 FDIP，BTB-X 在服务器 23 到服务器 32 的工作负载上也比 Conv-BTB 和 PDede 带来显著更高的收益。图10 还显示，FDIP 本身在 BTB 条目数量更多时表现更好。例如，在服务器 32 上，使用 Conv-BTB 的 FDIP 可带来 32% 的性能收益。使用 PDede 时，预取带来的性能收益增至 42%；使用 BTB-X 时则进一步增至 51%。

这些结果表明，在给定存储预算下容纳更多分支，BTB-X 不仅能减少流水线刷新，还能改善指令预取；这两者都能提升性能。

表 V：不同 BTB 设计的能耗需求。

BTB	访问类型	能耗（每次访问）	访问次数	能耗（总计）
Conv-BTB	Read	13.2pJ	1.60E+08	2122μJ
	写入	25.2pJ	4.36E+06	110μJ
	总能量			**2232μJ**
PDede	主 BTB 读取	8.4pJ	1.24E+08	1047μJ
	主 BTB 写入	12.5pJ	5.74E+05	7μJ
	页 BTB 读取	0.9pJ	2.01E+06	2μJ
	页 BTB 写入	0.8pJ	2.04E+04	0.02μJ
	页 BTB 搜索	6.2pJ	2.14E+05	2μJ
	总能量			**1058μJ**
BTB-X	Read	8.5pJ	1.16E+08	994μJ
	写入	11.4pJ	4.03E+05	5μJ
	总能量			**999μJ**

最后，图10显示，三种 BTB 组织方式在客户端工作负载上的表现相近。这是因为它们的分支工作集大多都能容纳在基线 Conv-BTB 中，而 PDede 和 BTB-X 增加的条目并未带来明显的性能提升。



(a) 服务器工作负载

(b) 客户端工作负载

图 11：在 0.9KB 存储预算下，传统 BTB、PDede 和 BTB-X 在 (a) **服务器** 和 (b) **客户端** 工作负载上的性能收益。X 轴表示 256、512、1K、2K、4K、8K 和 16K 条目 BTB-X 的存储需求。

E. 能耗与延迟分析

我们使用 Cacti 7.0 [35] 分析 Conv-BTB、PDede 和 BTB-X 在 22 纳米工艺节点下的能耗需求和访问延迟；这是 Cacti 支持的最新工艺节点。在此分析中，我们假设存储预算与第 VI-D 节性能分析中使用的相同，即 14.5KB。

能耗需求： 表 V 展示了不同 BTB 设计每次访问的读取和写入能耗需求。如表所示，BTB-X 与 PDede 的主分支目标缓冲器每次访问的读取和写入能耗非常相近。然而，除主分支目标缓冲器外，PDede 还需要为异页分支访问页分支目标缓冲器，即目标所在页与分支自身所在页不同的分支。此外，在 BTB 写入时，还需要搜索页分支目标缓冲器，以检查其中是否已存在目标页码。因此，对于异页分支，PDede 每次访问的读取和写入能耗分别达到 9.3 皮焦耳 (pJ) 和 19.5 pJ，而 BTB-X 分别为 8.5pJ 和 11.4 pJ。PDede 还包含区域分支目标缓冲区，但其能耗需求可以忽略不计，因此未在表 V 中列出。最后，Conv-BTB 每次访问的能耗显著高于 BTB-X，因为其每次读取和写入访问分别需要 13.2pJ 和 25.2pJ。

表 V 还展示了读写访问次数（按各工作负载取平均值）以及总能耗。尽管每次访问的能耗非常相近，PDede 的主分支目标缓冲器能耗却远高于 BTB-X。这是因为 PDede 的 MPKI 较高，常常会沿着错误的执行路径运行。表 V 中更高的分支目标缓冲器读取次数反映了这些额外的错误路径分支目标缓冲器访问，从而导致能耗增加。此外，由于 PDede 存储的分支较少，替换更加频繁，因此需要处理的分支目标缓冲器写入次数也比 BTB-X 更多。因此，PDede 的总能耗达到 1058μJ，而 BTB-X 为 999μJ。最后，Conv-BTB 的能耗需求为 2232μJ，显著高于 BTB-X，这是因为其每次访问能耗更高，且总访问次数更多。

总体而言，这项分析表明，BTB-X 不仅比 PDede 性能更好，而且能耗更低，因此能效也大幅提升。

访问延迟： 我们的分析表明，Conv-BTB 完成一次访问约需 0.36ns。如第 IV-B 节所述，PDede 的访问延迟是主分支目标缓冲器和页分支目标缓冲器访问延迟之和，因为这两种结构按顺序访问。我们的分析表明，访问主分支目标缓冲器和页分支目标缓冲器分别需要 0.34ns 和 0.13ns，因此 PDede 的总访问延迟为 0.47ns，明显高于 Conv-BTB。为解决这一问题，PDede 采用多周期分支目标缓冲器访问：第一个周期访问主分支目标缓冲器，只有当预测分支将被执行且其目标位于与该分支不同的页中时，才在下一个周期访问页分支目标缓冲器。因此，目标位于同一页的分支只需一个周期，而目标位于不同页且将被执行的分支则需要两个周期，才能从 PDede 获取其目标地址。最后，我们的分析表明，访问 BTB-X 仅需 0.33ns。总之，这项分析表明，BTB-X 在不影响访问延迟的情况下，提供了更高的存储效率。

F. 不同分支目标缓冲器存储预算下的性能变化

为进一步了解 BTB-X 相较于 PDede 和 Conv-BTB 的性能优势，我们比较了它们在不同存储预算下的性能。图 11 展示了服务器和客户端工作负载上的性能提升。结果以存储预算为 0.9KB 的 Conv-BTB 性能为基准进行了归一化。所有设计（包括基准设计）均启用了指令预取。

如图所示，在服务器工作负载上，当存储预算分别不超过 29KB 和 14.5KB 时，BTB-X 的性能显著优于 Conv-BTB 和 PDede。BTB-X 的性能优势

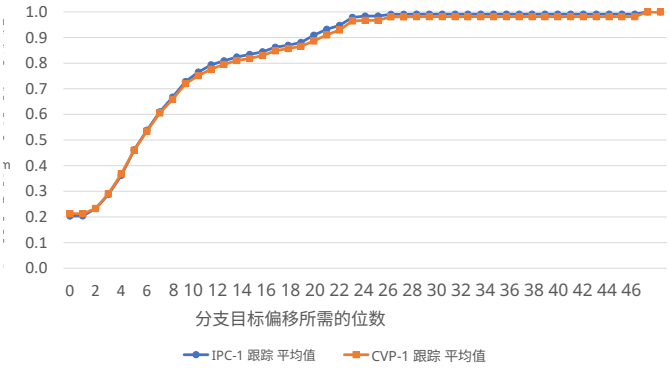


图 12：CVP-1 和 IPC 跟踪中的目标偏移分布。

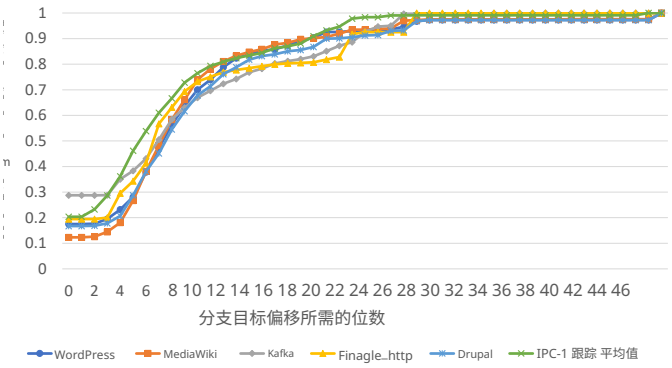


图 13：x86 编译的服务器应用程序和 Arm64 每周期指令数跟踪中的目标偏移分布。

在指令占用空间较大、给 BTB 和一级指令缓存带来压力的服务器跟踪上尤为明显。例如，在 14.5KB 的存储预算下，BTB-X 相较于基准的性能收益为 35%，而 PDede 和 Conv-BTB 分别为 29% 和 20%。当 BTB 存储空间预算较大时，许多工作负载的分支工作集开始能够容纳在可用的 BTB 容量中，此时 BTB-X 与另外两种设计之间的性能差距便会缩小。此外，由于客户端跟踪的指令工作集较小，三种 BTB 组织结构之间的性能差距会更早趋于平稳。

从该图中可以得出的一个重要结论是，即使 BTB-X 的存储预算仅为传统 BTB 的一半，其性能仍优于传统 BTB。例如，在图 11a 中，存储预算为 14.5KB 的 Conv-BTB 可提升 20% 的性能，而 BTB-X 仅用 7.25KB 就能提升 24%。这一现象的原因是，在存储预算相同的情况下，BTB-X 可容纳的条目数量是 Conv-BTB 的 2.24 倍；因此，即使将 BTB-X 的预算减半，其容量仍略胜于 Conv-BTB。

G. 分析更多工作负载中的目标偏移分布

我们研究了 750+高通为首届冠军价值预测竞赛（CVP-1） [6]提供的服务器轨迹中的目标偏移分布。图 12 所示结果表明，其偏移分布与图 4 所示的 IPC-1 跟踪中的分布非常相似。这项研究证实，这种偏移分布是由应用程序的编写方式及其产生的控制流行为所决定的。如第二节所述，这种偏移分布源于条件分支在动态分支工作集中占主导地位，且其偏移量通常较短。这是因为条件分支引导函数内部的控制流，而软件工程原则倾向于采用较小的函数。因此，短偏移量在分支偏移量分布中占主导地位。

除了 CVP-1 跟踪，我们还分析了另外五种服务器应用：来自 Facebook HHVM 开源软件性能基准测试的 WordPress [53]，MediaWiki [52]，和 Drupal [51]、来自 Java DaCapo 的 Kafka [8]，以及来自 Java Renaissance 的 Finagle-HTTP [50]。此外，这些应用被编译为 x86（CVP-1 和 IPC-1 跟踪被编译为 Arm64），这也使我们能够评估指令集架构对目标偏移分布的影响。图 13 展示的结果表明，这些应用中的偏移分布也与 IPC-1 跟踪中的非常相似。唯一的区别是，x86 跟踪需要稍大的偏移量（多 1 或 2 位），才能达到与 Arm64 （CVP-1 和 IPC-1）跟踪相似的动态分支覆盖率。例如，6 位偏移量可覆盖 Arm64 跟踪中约 54% 的分支，而 x86 偏移量则需要 8 位才能达到 58% 的分支覆盖率。这是因为 x86 偏移量以字节 为单位指定分支 PC 与目标之间的距离，因为 x86 指令的长度可变。相比之下，Arm64 偏移量以指令为单位指定这一距离，因为所有指令的长度都是 4 字节，因此可节省 2 位偏移量。

由于 BTB-X 需要为 x86 存储比 Arm64 稍大的偏移量，我们重新评估了其相较于 PDede 和 Conv-BTB 在 x86 架构上的存储优势。由于 8 路 BTB-X 中的每一路都需要覆盖约 12.5% 的分支，因此我们根据图 13 所示的 x86 应用程序偏移分布，将各路配置为存储 0、5、6、7、9、12、20 和 27 位偏移量。因此，每组需要 86 位来存储偏移量，而 Arm64 中为 80 位。因此，BTB-X 在 x86 上的存储优势略低于 Arm64。不过，对于 x86，BTB-X 存储的分支数仍是 Conv-BTB 的 2.18 倍（Arm 上为 2.24 倍）。与 PDede 相比，在 0.9KB 存储预算下，BTB-X 存储的分支数多 1.21 倍（Arm64 上为 1.24 倍）；在 58KB 存储预算下，则多 1.31 倍（Arm64 上为 1.34 倍）。（第VI-B节介绍了对 Arm64 跟踪的这项分析。）

VII. 相关工作

缓解BTB未命中： Losq [36] 最早披露了分支目标缓冲器，随后 Lee等人 [33]对其进行了扩展。由于分支目标缓冲器位于指令传送的关键路径上，因此已有多种方案旨在提升其效能。Yeh等人 [54] 提出，不再使用每条指令各自的程序计数器访问分支目标缓冲器，而是使用基本块地址进行访问，并在分支目标缓冲器中同时存储目标地址和顺序执行地址。如果预测分支不跳转，那么在取完当前基本块后，顺序执行地址将用作下一条指令取指以及下一次分支目标缓冲器访问的程序计数器，

作为下一次分支目标缓冲器访问的程序计数器。与传统BTB组织结构相比，这种 BTB组织结构的优势在于能够降低分支目标缓冲器带宽和功耗需求，因为一次访问即可提供下一个控制流分歧点，而传统组织结构需要访问的次数与到达下一条分支之前的指令数相同。最初的基于基本块的分支目标缓冲器方案 [54] 存储完整的顺序执行地址，而后续工作 [44] 提出存储两个连续基本块地址之间的差值。Fagin [18] 提出通过仅存储部分标签，更有效地利用分支目标缓冲器存储空间。为了进一步分摊标签的存储开销，一些设计提出让位于同一缓存块中的多个分支共享一个分支目标缓冲器条目 [2], [27]。尽管这些 BTB 设计旨在改进 BTB 管理的不同方面，但它们都有一个共同特点，即存储完整目标地址。因此，BTB-X 的关键思想可应用于所有这些 BTB 设计，以降低其目标地址存储开销。

先前的研究 [10], [11], [23], [47]–[49]也探索了降低分支目标存储开销的机制。Seznec [47], [48] 提出将目标地址拆分为页码和偏移量；在 BTB 中存储指向页码的指针以及页偏移，而页码本身则存储在单独的结构中。由于指向页码的指针比页码本身更小，而且同一页中所有目标的页码只需存储一次，因此这种方法降低了存储开销。Hoogerbrugge [23] 提出在一个组中配置部分条目来存储较小的目标偏移量，从而降低 BTB 的存储需求。

最先进的 BTB 设计 PDede [49],结合了这两种思路，以克服各自的局限。具体而言，对于同页分支，Seznec 的设计并非最优，因为它会不必要地存储其目标页码（的指针），尽管该页码与其分支程序计数器的页码相同。相比之下，对于跨页分支，Hoogerbrugge 的设计并非最优，因为它存储的是完整目标地址。受 Hoogerbrugge 设计的启发，PDede 将一个组中的部分条目配置为存储同页分支目标；对于跨页分支，它则类似于 Seznec 的设计，存储指向页码的指针，而不是页码本身。PDede 还通过将页码拆分为页码和区域编号，进一步降低了跨页目标的存储开销。然而，由于它基于 Seznec 的设计，因此也必须承担主分支目标缓冲器与页/区域分支目标缓冲器之间间接寻址带来的额外延迟开销。微型分支目标缓冲区 [22], 提出了一种灵活的 BTB 条目结构，每个条目可存储一个偏移量较大的分支，或两个偏移量较小的分支。我们表明，所有这些设计都未能充分利用分支偏移量分布不均所带来的存储优化机会。BTB-X 不仅抓住了这一机会，还避免了最先进设计中的 BTB 间接寻址。

除了优化 BTB组织结构外，先前的工作 [15], [16], [28], [31], [32] 还探索了 BTB 预填充/预取，以缓解 BTB 未命中。当前最先进的分支目标缓冲区预取技术是一种基于剖面引导的软件预取器，名为 Twig [28]。它会分析应用程序的执行剖面，以识别关键的 BTB 未命中，然后注入软件预取指令。预取指令以压缩分支程序计数器和目标作为操作数，其执行会将这些信息填入 BTB。这些预取技术与 BTB组织结构互为补充，因此可以与 BTB-X 一起使用。**缓解一级指令缓存缺失：**随着一级指令缓存缺失持续成为服务器应用程序的主要性能瓶颈， [12], [25],[46],先前的工作提出了硬件和软件机制来缓解一级指令缓存缺失。在硬件方面，当前最先进的时序流预取器 [19], [20] 会记录一级指令缓存缺失/访问历史，并重放这些历史以发现预取候选项。尽管这类预取器非常有效，但其巨大的元数据存储成本使其不切实际，尽管近期已有研究尝试解决这一弱点 [26], [27]。取指引导预取器利用核内结构（BTB 和分支方向预测器）超前于取指单元运行，以寻找预取候选项。早期工作 [43]仅专注于一级指令缓存预取，而当前最先进的取指引导预取器[31], [32]还会将内容预填充到 BTB 中。

此外，还有一些纯软件方法被提出，用于指令预取和提升一级指令缓存容量 [9], [13], [17], [29], [34], [37]–[40]。这些方法利用应用程序剖析数据，进行编译时、链接时或链接后优化。由于这些方法完全基于软件，因此也能受益于 BTB-X 组织结构所提供的更大分支目标缓冲器容量。

VIII. 结论

当代服务器应用程序拥有数兆字节的指令占用空间，导致分支目标缓冲器和一级指令缓存频繁缺失，已成为主要的性能瓶颈。分支目标缓冲器容量会影响流水线刷新率和取指驱动的指令预取效果，进而极大地影响前端性能，因此商业产品会为分支目标缓冲器分配数十至数百 KB 的存储空间。我们发现，分支目标缓冲器存储开销中占比最大的部分是存储分支目标的开销。我们还发现，存储目标偏移量而非完整目标地址甚至压缩目标，可以大幅降低分支目标缓冲器存储开销。这是因为大多数分支的目标都离分支本身相对较近，而且我们的分析表明，超过 99% 的偏移量所需的位数至多为存储完整目标地址所需位数的一半。基于这些观察，我们提出了一种存储效率高的 BTB 组织结构，称为 BTB-X，它用目标偏移量代替目标地址进行存储。此外，BTB-X 是一种八路组相联分支目标缓冲区，采用大小不同的路，每一路存储不同长度的偏移量，从而适应偏移量长度分布不均的情况。总体而言，在相同存储预算下，BTB-X 能够存储的分支数量约为传统 BTB 的 2.24 倍，也达到最先进 BTB 组织结构的 1.3 倍。

致谢

本研究部分由挪威研究委员会（NFR）向 NTNU 提供的302279号资助支持。

参考文献

[1] “第1届指令预取竞赛”，https://research.ece.ncsu. edu/ipc/. [2] “AMD 软件优化指南。第 2.8.1.2 节”，https://www.amd.com/system/files/TechDocs/56665.zip. [3] “BTB-X GitHub Ripository”，https://github.com/rakeshdhakla/ ChampSim-master-BTBX. [4] “ChampSim模拟器”，https://github.com/ChampSim/ChampSim.[5] “探究英特尔 Ice Lake处理器：一窥 Sunny Cove微架构”，https://www.anandtech.com/show/14514/examining-intels-ice-lake-microarchitecture-and-sunny-cove/3. [6] “首届值预测竞赛”，https://www.microarch.org/ cvp1/cvp1online/contestants.html. [7] “Twitter Finagle”，https://twitter.github.io/finagle/. [8] “facebookarchive/oss-performance：用于基准测试各种 PHP 实现运行开源软件的脚本”，https://github.com/facebookarchive/oss-performance, 2022. [9] M. Annavaram、J. M. Patel 和 E. S. Davidson，“数据库应用的调用图预取”，ACM计算机系统汇刊（TOCS），第 21 卷，第 4 期，第 412–444 页，2003年。 [10] T. Asheim、B. Grot 和 R. Kumar，“BTB-X：高存储效率的分支目标缓冲区组织结构”，IEEE计算机体系结构快报，第 20 卷，第 2 期，第 134–137 页，2021年。[11] T. Asheim、B. Grot 和 R. Kumar，“面向服务器的专用分支目标缓冲器组织结构”，载于第 31 届并行体系结构与编译技术国际会议（PACT）论文集，2022年。[12] T. Asheim、T. A. Khan、B. Kasicki 和 R. Kumar，“微体系结构状态复用对无服务器函数的影响”，载于第八届无服务器计算国际研讨会论文集，丛书 WoSC ’22。美国纽约：计算机协会，2022年，第 7 12 [ ] 10 1145 3565382 3565879 – 页。[13] 在线。可获取于：https://doi.org/ . / . G. Ayers、N. P. Nagendra、D. I. August、H. K. Cho、S. Kanev、C. Kozyrakis、T. Krishnamurthy、H. Litz、T. Moseley 和 P. Ran- ganathan，“Asmdb：理解并缓解仓库级计算机中的前端停顿”，载于第 46 届国际计算机体系结构研讨会论文集，2019年，第 462–473 页。[14] S. M. Blackburn、R. Garner、C. Hoffmann、A. M. Khang、K. S. McKinley、R. Bentzur、A. Diwan、D. Feinberg、D. Frampton、S. Z. Guyer 等，“DaCapo基准测试：Java 基准测试的开发与分析”，载于第 21 届 ACM SIGPLAN面向对象编程系统、语言与应用会议论文集，2006年，第 169–190 页。[15] J. Bonanno、A. Collura、D. Lipetz、U. Mayer、B. Prasky 和 A. Saporito，“双级批量预加载分支预测”，载于国际高性能计算机体系结构研讨会，2013年，第 71–82 页。[16] I. Burcea 和 A. Moshovos，“Phantom-BTB：虚拟化分支目标缓冲区设计”，载于第 14 届编程语言与操作系统体系结构支持国际会议论文集（ASPLOS 2009），美国华盛顿哥伦比亚特区，2009年3月7日至11日，2009年，第 313–324 页。 [在线]。可获取于：http: //doi.acm.org/10 1145 1508244 1508281 。/。D. Chen、T. Moseley 和 D. X. Li，“AutoFDO：面向仓库级应用的自动反馈引导优化”，载于 2016 IEEE/ACM 代码生成与优化国际研讨会（CGO）。IEEE，2016年，第 12–23 页。[18] B. Fagin，“分支目标缓冲区中的部分解析”，IEEE计算机汇刊，第 46 卷，第 10 期，第 1142–1145 页，1997年。[19] M. Ferdman 等，“时序指令获取流”，载于国际微体系结构研讨会，2008年。[20] M. Ferdman 等，“主动指令取指”，载于国际微体系结构研讨会，2011年。[21] B. Grayson、J. Rupley、G. Z. Zuraski、E. Quinnell、D. A. Jim´enez、T. Nakra、P. Kitchin、R. Hensley、E. Brekelbaum、V. Sinha 和 A. Ghiya，“三星Exynos CPU微架构的演进”，载于 2020 年 ACM/IEEE 第 47 届年度国际计算机体系结构研讨会（ISCA），2020年，第 40–51 页。[22] V. Gupta 和 B. Panda，“微型分支目标缓冲区：面向服务器的高性能、高存储效率未级分支目标缓冲区”，载于第 19 届 ACM计算前沿国际会议论文集，丛书 CF ’22。美国纽约：计算机协会，2022年，第 12–20 页。 [在线]。可获取于：https://doi.org/10.1145/3528416.3530224 [23] J. Hoogerbrugge，《高性价比分支目标缓冲器》，载于 Euro-Par 2000 并行处理。柏林，海德堡：施普林格柏林海德堡，2000年，第950–959页。[24] Y. Ishii、J. Lee、K. Nathella 和 D. Sunwoo，《重新确立取指引导的指令预取：行业视角》，载于2021年IEEE系统与软件性能分析国际研讨会（ISPASS），2021年，第172–182页。[25] S. Kanev、J. P. Darago、K. Hazelwood、P. Ranganathan、T. Moseley、G.-Y. Wei 和 D. Brooks，《仓库规模计算机的性能剖析》，载于第42届计算机体系结构年度国际研讨会论文集，系列：ISCA ’15。美国纽约：计算机协会，2015年，第158–169 页。 [在线]。链接：https://doi.org/10 1145 2749469 2750392 。/。C. Kaynak 等，《SHIFT：面向精简核心服务器处理器的共享历史指令获取》，载于国际微体系结构研讨会，2013年。[27] C. Kaynak、B. Grot 和 B. Falsafi，《Confluence：面向横向扩展服务器的统一指令供给》，载于2015年IEEE/ACM第48届微体系结构年度国际研讨会（MICRO），2015年，第166–177页。[28] T. A. Khan、N. Brown、A. Sriraman、N. K. Soundararajan、R. Kumar、J. Devietti、S. Subramoney、G. A. Pokam、H. Litz 和 B. Kasikci，《Twig：面向数据中心应用的基于性能剖析引导的分支目标缓冲区预取》，载于 MICRO-54：第54届IEEE/ACM微体系结构年度国际研讨会，系列：MICRO ’21。美国纽约：计算机协会，2021年，第816–829页。 [在线]。链接：https://doi.org/10 1145 3466752 3480124 。/。T. A. Khan、A. Sriraman、J. Devietti、G. Pokam、H. Litz 和 B. Kasikci，《I-spy：通过合并实现的上下文驱动条件指令预取》，载于2020年IEEE/ACM第53届微体系结构年度国际研讨会（MICRO）。IEEE，2020年，第146–159页。[30] R. Kumar 和 B. Grot，《消除服务器前端瓶颈》，ACM计算机系统汇刊，第38卷，第3–4期，2022年1月。 [在线]。链接：https://doi.org/10 1145 3484492 。/。R. Kumar、B. Grot 和 V. Nagarajan，《借助 shotgun 突破前端瓶颈》，载于第二十三届编程语言与操作系统体系结构支持国际会议论文集，系列：ASPLOS ’18。美国纽约：计算机协会，2018年，第30–42页。 [在线]。链接：https://doi.org/10 1145 3173162 3173178 。/。R. Kumar、C.-C. Huang、B. Grot 和 V. Nagarajan，《Boomerang：用于控制流传送的无元数据架构》，载于2017年IEEE高性能计算机体系结构国际研讨会（HPCA），2017年，第493–504页。[33] Lee 和 Smith，《分支预测策略与分支目标缓冲区设计》，Computer，第17卷，第1期，第6–22页，1984年。[34] D. X. Li、R. Ashok 和 R. Hundt，《轻量级反馈引导的跨模块优化》，载于第8届IEEE/ACM代码生成与优化年度国际研讨会论文集，2010年，第53–61页。[35] S. Li、K. Chen、J. H. Ahn、J. B. Brockman 和 N. P. Jouppi，《Cacti-p：采用先进漏电降低技术的基于SRAM的结构的体系结构级建模》，载于2011年IEEE/ACM计算机辅助设计国际会议（ICCAD），2011年，第694–701页。[36] J. Losq，《用于分支预测的通用历史表（适用于流水线计算机）》，IBM技术披露公报，第1卷，1982年。[37] C.-K. Luk、R. Muth、H. Patil、R. Cohn 和 G. Lowney，《Ispike：用于Intel/spl reg/Itanium/spl reg/体系结构的链接后优化器》，载于代码生成与优化国际研讨会，2004年。CGO 2004。IEEE，2004年，第15–26页。[38] C.-K. Luk 和 T. C. Mowry，《协同预取：编译器和硬件支持现代处理器实现有效的指令预取》，载于第31届ACM/IEEE微体系结构年度国际研讨会论文集。IEEE，1998年，第182–193页。[39] G. Ottoni 和 B. Maher，《面向大规模数据中心应用的函数布局优化》，载于2017年IEEE/ACM代码生成与优化国际研讨会（CGO）。IEEE，2017年，第233–244页。[40] M. Panchenko、R. Auler、B. Nell 和 G. Ottoni，《Bolt：面向数据中心及更广泛领域的实用二进制优化器》，载于2019年IEEE/ACM代码生成与优化国际研讨会（CGO）。IEEE，2019年，第2–14页。[41] A. Pellegrini、N. Stephens、M. Bruce、Y. Ishii、J. Pusdesris、A. Raja、C. Abernathy、J. Koppanalil、T. Ringe、A. Tummala、J. Jalal、M. Werkheiser 和 A. Kona，《Arm Neoverse N1平台：构建

面向下一代云到边缘基础设施SoC的模块》，*IEEE Micro*，第40卷，第2期，第53–62页，2020年。[42] A. Prokopec、A. Ros`a、D. Leopoldseder、G. Duboscq、P. T˚uma、M. Stu- dener、L. Bulej、Y. Zheng、A. Villaz´on、D. Simon、T. W¨urthinger 和 W. Binder，《Renaissance：面向Java虚拟机并行应用的基准测试套件》，载于编程语言设计与实现，2019年。[43] G. Reinman、B. Calder 和 T. Austin，《取指引导的指令预取》，载于*MICRO-32*。第*32*届*ACM/IEEE*微体系结构年度国际研讨会论文集，1999年，第16–27页。[44] G. Reinman、T. Austin 和 B. Calder，《用于快速指令传送的可扩展前端架构》，载于第*26*届计算机体系结构年度国际研讨会论文集，系列：ISCA '99。美国：IEEE计算机学会，1999年，第234–245页。 [在线]。链接：https://doi.org/10 1145 300979 300999 。/。A. Saporito，《IBM z15 处理器芯片组》，载于*Hot Chips*，2020年。[46] D. Schall、A. Margaritov、D. Ustiugov、A. Sandberg 和 B. Grot，《温热的无服务器函数：特征分析与优化》，载于第*49*届计算机体系结构年度国际研讨会论文集，系列：ISCA '22。美国纽约：计算机协会，2022年，第757–770页。 [在线]。链接：https://doi.org/10 1145 3470496 3527390 。/。A. Seznec，《不要使用页码，而应使用指向该页的指针》，载于第*23*届计算机体系结构年度国际研讨会论文集，系列：ISCA '96。美国纽约：计算机协会，1996年，第104–113页。 [在线]。链接：https://doi.org/10 1145 232973 232985 。/。A. Seznec，《-k字节 ITTAGE 间接分支预测器》，*J. Instruction- Level Parallelism*，2011年。[49] N. K. Soundararajan、P. Braun、T. A. Khan、B. Kasikci、H. Litz 和 S. Subramoney，《PDede：分区、去重和差分分支目标缓冲区》，载于*MICRO-54*：第*54*届*IEEE/ACM*微体系结构年度国际研讨会，系列：MICRO '21。美国纽约：计算机协会，2021年，第779–791页。 [ ] 10 1145 3466752 3480046 在线。链接：https://doi.org/。/。维基百科贡献者，《Apache Kafka——维基百科，自由的百科全书》，https://en.wikipedia.org/w/index.php?title=Apache Kafka&oldid=988898935，2022。[51] 维基百科贡献者，《Drupal——维基百科，自由的百科全书》，https://en.wikipedia.org/w/index.php?title=Drupal&oldid=989582664，2022年。[52] 维基百科贡献者，《Mediawiki——维基百科，自由的百科全书》，https://en.wikipedia.org/w/index.php?title=MediaWiki& oldid=989993176，2022年。[53] 维基百科贡献者，《Wordpress——维基百科，自由的百科全书》，https://en.wikipedia.org/w/index.php?title=WordPress& oldid=977243718，2022年。[54] T.-Y. Yeh 和 Y. N. Patt，《支持推测执行的处理器综合指令取指机制》，载于第*25*届微体系结构年度国际研讨会论文集，系列：MICRO 25。美国华盛顿哥伦比亚特区：IEEE计算机学会出版社，1992年，第129–139页。

附录 A：制品附录

A. 摘要

我们在 Champsim 模拟器中实现了 BTB-X。我们的制品包括：1）ChampSim 中的 BTB-X 实现；2）工作负载轨迹链接；3）用于生成配置文件、启动模拟和收集结果的脚本；4）用于绘制最重要结果的 Excel 文件。我们确定了制品评估的三项关键结果：a）分支目标偏移分布（图 4）；b）BTB MPKI 降低（图 9）；c）性能提升（图10）。

B. 元信息

- **编译：** 使用 GCC 8.5.0 测试。它也应该适用于其他较新版本的 GCC。
- **代码/工作负载：** 请从提供的链接下载代码/工作负载。
- **实验：** 按照下文所述修改提供的脚本，以运行模拟。
- **指标：** 每周期指令数、BTB 每千条指令未命中数、分支目标偏移分布。
- **运行实验所需时间：** 所有跟踪并行运行时，不到 30 分钟。
- **绘制图表：** 提供了 Excel 文件 *BTBX artifact results.xlsx*，用于绘制图表。

C. 获取工件

- **代码：** 从 [3] 下载 BTB-X 实现。
- **工作负载：** 可从 https://drive.google.com/file/d/1qs8t8-YWc7lLoYbjbHd3lf1xdoYBznf/view?usp=sharing 下载工作负载。将工作负载放入 <代码路径*/dpc3 traces/*。
- **Excel 文件：** 要绘制图表，请下载 Excel 文件 *BTBX artifact results.xlsx*，来自 [3]。

D. 系统要求

任何能够运行 Champsim 的硬件都足够。建议使用 SLURM 在集群上运行仿真。这些脚本使用 Bash 编写。

E. 实验工作流程

- **编译：** Champsim 需要使用三种 BTB 设计（convBTB、pdede 和 BTBX）以及两种指令预取器（no、fdip）进行编译。按照 [3] 中的说明编译代码。编译重要提示：使用“fdip”预取器编译时，IFETCH BUFFER 需要设置为 128 个条目；使用“no”预取器编译时，则需要设置为“FETCH WIDTH*2”个条目。这是因为基准版 Champsim 中指令取指的实现方式所致。IFETCH BUFFER 的大小在<代码路径*>/inc/ooo_cpu.h* 的第 63 行定义。
- **生成配置文件：** 转到目录<代码路径*>/launch/scripts/*。在脚本文件*createConfig.sh*中，将 Champsim 路径设置为<代码路径>。运行此脚本（*./createConfig.sh*）以生成 Champsim 所需的配置文件。
- **运行模拟：**
  运行所有工作负载：前往目录<代码路径*>/launch/*。在脚本文件*launch.sh*中，将第 64 行的 <此处填写集群启动命令>替换为在你的集群上运行实验的命令。此处提供了一个示例命令，用于在我们的集群上运行实验。运行此脚本（*./launch.sh*）将执行仿真，并将统计数据存储在目录<代码路径*>/results_50M/*中。运行单个工作负载：此处[3]提供了一个用于运行单个工作负载仿真的示例命令。

F. 结果

- **收集结果：** 前往 <代码路径>*/collectStats/*。运行脚本 *getResults.sh*，它将收集所有工作负载的结果，并将其保存到文件 *all res*中。
- **绘制结果：** 下载*all res* 文件。打开提供的 Excel 文件 *BTBX artifact results.xlsx*。点击 MS-Excel 顶部菜单栏中的“数据”。在“查询和连接”功能区中点击“全部刷新”，然后前往存放*all res* 的文件夹，并双击*all res*。现在，Excel 文件中的“偏移分布”、“MPKI”和“性能”工作表中应分别包含图 4、图 9 和图10的图表。