doi:10.3772/j.issn.1002-0470.2024.07.001

基于动态压缩的高存储效率末级分支目标缓冲①

谭弘泽② 王 剑

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

(中国科学院大学 北京 100049)

摘 要 随着软件系统规模及复杂度的增长,数量庞大的指令使指令高速缓存和分支目标缓冲(BTB)频繁地发生缺失,导致中央处理器(CPU)性能下降。现代工业 CPU 设计在分离式前端中使用充分大的多级 BTB 以减少缺失导致的性能损失。由于实际芯片的存储资源有限,大容量的末级 BTB 需要更高的存储效率。然而,现有压缩 BTB 采用静态分配目标偏移量存储空间的方法,无法按照分支的实际存储需求进行调整,导致其存储效率较低。针对上述问题,提出一种基于动态压缩的 BTB——ZBTB。ZBTB 通过可变长编码表示目标偏移量,动态分配目标偏移量存储空间,结合无额外存储的最近最少使用(LRU)和偏斜相联等方法缓解冲突,提升了存储效率。基于以第 1 届指令预取锦标赛(IPC-1)所发布轨迹数据进行的评估,与现有 BTB 相比,ZBTB 在 33.5 kB 容量下可将误预测次数降低 66%。

关键词 分支预测;分支目标缓冲(BTB);压缩;偏斜相联

中央处理器(central processing unit,CPU)可被分为前端和后端 2 部分,其中前端负责取出新的指令,而后端负责执行所取出的指令。为了挖掘指令级并行性、提高 CPU 性能,前端使用指令高速缓存和分支目标缓冲(branch target buffer, BTB)缓存频繁使用的指令和分支,从而缓解为后端提供指令的延迟。

然而,现代软件系统不断增长的复杂性为处理器前端带来了越来越大的压力[1]。除此之外,为了跨指令系统兼容性,Java 等语言中广泛使用的即时(just-in-time,JIT)编译及二进制翻译[2]等技术还会进一步增加指令数量,从而对处理器前端产生更大压力。当指令和分支数量分别超出 CPU 指令高速缓存和 BTB 的容量时,这些指令和分支就会发生缺失,从而大幅降低 CPU 性能。

取指导向预取(fetch-directed prefetching,FDP)技术[3]可以通过分支预测降低因指令高速缓存缺失带来的性能损失,将指令高速缓存的存储压力转移到分支预测器部分。工业界使用可容纳高达1.28×10^5条分支的 BTB[4],在指令高速缓存缺失时保障分支预测。

在容量有限的情况下,现有 BTB 通过替换、压缩和预取等算法来降低缺失导致的性能损失,并采用组相联或偏斜相联[5]的存储结构减少冲突。其中,随机、最近最少使用(least recently used,LRU)和重用距离预测(re-reference interval prediction,RRIP)[6]等替换算法通过对逐出表项的选择保留最可能命中的表项。分割去重差分分支目标缓冲(partitioned deduplicated delta BTB,PDeDe)[7]、BTB-X[8]和 MB-TB[9]等 BTB 设计采用压缩算法降低表项的存储开销,从而通过增加可存储的表项数目减少缺失。文献[10]提出的预取算法通过提前向 BTB 装入分支

① 国家重点研发计划(2022YFB3105103)资助项目。

② 男,1996 年生,博士生;研究方向:计算机系统结构,CPU 设计;联系人,E-mail:tanhongze20b@ict.ac.cn。

(收稿日期:2023-05-17)

信息避免缺失。CPU 设计可在高存储效率 BTB 的基础上使用预取算法进一步降低缺失导致的性能损失。

为了进一步提高 BTB 的存储效率,本文提出了一种动态压缩的 BTB——ZBTB(zipped branch target buffer)。ZBTB 通过可变长编码表示目标偏移量,动态分配目标偏移量存储空间,结合无额外存储的最近最少使用(LRU)和偏斜相联等方法缓解冲突,提升存储效率。

本文的主要贡献有以下 3 个方面。

(1) 分析了目标地址存储的冗余,提出了一种动态空间分配的目标地址压缩方法。

(2) 在所提动态目标地址压缩方法的基础上,结合偏斜相联及替换算法等提高存储效率的方法,提出了一种可实现的高存储效率 BTB——ZBTB。

(3) 基于第 1 届指令预取锦标赛(the 1st Instruction Prefetching Championship, IPC-1)[11]所发布轨迹数据对 ZBTB 进行了验证及对比,相比现有 BTB,实现了 66% 的误预测次数降低。

1　研究背景

1.1　流水线结构

分离式前端[3,12]被工业界[4]和学术界[13]广泛应用,其结构如图 1 所示。其中,分支预测单元(branch predictionunit,BPU)将分支目标缓冲(BTB)与条件分支预测器、间接跳转预测器和返回地址栈(return address stack,RAS) 3 种预测器相组合[14],预测控制流并缓存至取指目标队列(fetch target queue,FTQ)。取指单元(instruction fetch unit,IFU)可根据 FTQ 中提前预测的控制流信息进行取指或 FDP,从而缓解处理器的前端阻塞。理想的 BTB 可为处理器提升约 20% 的性能[9]。

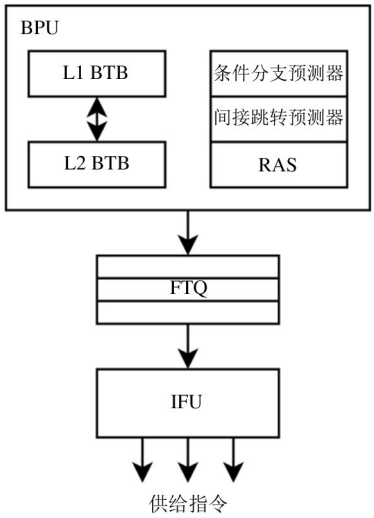

图 1　分离式前端结构

为了兼顾访问延迟和存储容量,BPU 可采用多级 BTB 结构。其中,L1 BTB 延迟短而容量小,L2 BTB 容量更大但延迟也更大。L1 BTB 命中可以不必再访问 L2 BTB,从而掩盖 L2 BTB 的延迟和读取功耗。而在 L1 BTB 缺失时,L2 BTB 查询的结果可以填回 L1 BTB 来弥补容量的不足。

在多级 BTB 结构中,由于末级 BTB 仅需在其他层级 BTB 均缺失的少量情况下进行读取,末级 BTB 具有容量最大而访问次数最少的特点,允许采用更激进的方式提升存储效率。其巨大的容量使得存储效率的提升能够更有效地降低 BTB 的总容量,而稀少的访问次数则缓解了访问延迟对性能的影响[9]。此外,末级 BTB 可与指令高速缓存并行访问,从而避免增加流水线深度,进一步降低访问延迟对性能的影响。

1.2　查询方式

BTB 可以选择按指令地址或按取指块[12] 2 种查询方式。其中,传统 BTB 需要按控制流各个指令地址依次查询。取指目标缓冲(fetch target buffer,FTB)[12]按取指块查询,可以减少查询次数、提高取指带宽和设计主频[13],但存在重复存储的问题。通过在 FTB 项中存储多条分支的相关信息,多分支预测[15]可以进一步提升预测带宽。通过沿可能的跳转路径向查询后续分支,多分支的 FTB 结构也可采用单分支的末级 FTB。

相比于按指令地址的查询方式,图2展示了按取指块查询的存储开销在 IPC-1 不同负载中的增加。在存储压力较大的客户端负载及服务器负载中,按取指块的查询仅增加 2% ~4% 的存储开销。

本文以按指令地址的查询方式介绍设计并进行评估和对比。通过改变查询方式并在表项中加入基本块长度信息,相似的设计可以用于 FTB 结构。

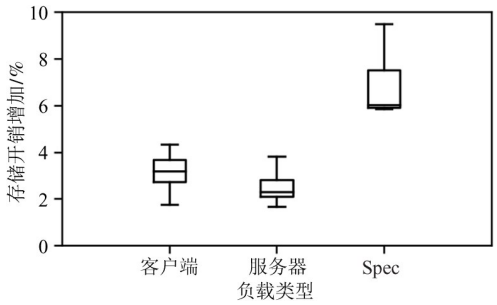

图 2　不同负载下按取指块查询的存储开销增加

1.3　存储内容

传统 BTB 采用组相联结构,表项包含有效位、标签、分支类型、替换信息和目标地址 5 部分,其结构及其典型位宽如图 3 所示。其中,目标地址位数可达 46,标签位数为 12 ~ 16。

BTB 可用 3 位二进制数编码 6 种分支类型,其所编码分支类型及与其他预测器组合方式如表 1 所示。其中,BTB 需要为条件分支、直接跳转和直接函数调用 3 种直接分支提供跳转地址,不需要为函数返回、间接跳转和间接函数调用 3 种间接分支提供跳转地址。特别地,为了减轻条件分支预测器和间接跳转预测器的负担,BTB 可忽略不跳转的条件分支,将一直跳转的条件分支和固定目标地址的间接跳转视为直接跳转,并将固定目标地址的间接函数调用视为直接函数调用。

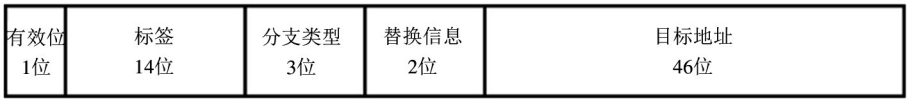

图 3　传统 BTB 表项结构及典型位宽

表 1　BTB 分支类型及与其他预测器组合方式

分支类型	跳转方向来源	跳转地址来源
条件分支	条件分支预测器	BTB
函数返回	无	RAS
直接跳转	无	BTB
直接函数调用	无	BTB
间接跳转	无	间接跳转预测器
间接函数调用	无	间接跳转预测器

1.4　标签压缩

现有 BTB 通过哈希将完整查询地址映射为位数较少的哈希值标签,以允许少量哈希混淆为代价,实现了标签压缩。其中,哈希算法通过压缩映射将完整地址变换成较短的哈希值。由于不同的哈希值代表不同的地址,哈希值可作为查询依据区分不同地址。由于压缩映射可能将不同地址映射为相同的哈希值,这些位于不同地址的分支会使用相同的哈希值作为标签,从而导致 BTB 产生错误预测。这种不同地址使用相同哈希值而导致混淆的问题被称为哈希混淆。

采用组相联结构的传统 BTB 以地址低位为索引,并以地址高位哈希值作为标签进行查询。以 IPC-1 数据集[11] 上 8 路 512 组共 4 096 项的传统 BTB 为例,不同标签位数下的每千指令混淆次数如图 4 所示。在标签位数降低到 14 位以下时,BTB 将产生无法忽略的哈希混淆。

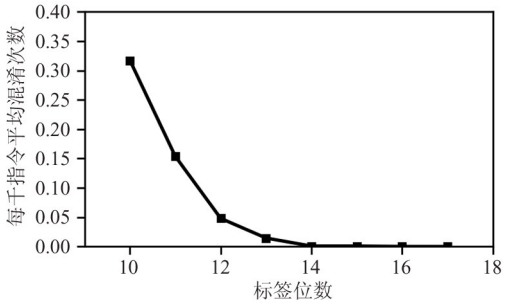

图 4　不同标签位数下的混淆率变化

2　相关工作

本节从存储冗余消除和存储格式设计 2 方面介绍相关工作,并对部分现有 BTB 提出修正。

2.1　存储冗余消除

现有 BTB 尝试从目标地址间的相关性和程序的空间局部性 2 种角度消除存储冗余。其中,目标地址间的相关性来自于不同目标地址高位经常相同的特点[16]。而程序的空间局部性则使得大多数目标地址具有指令地址相同的高位。

PDeDe[7] 利用目标地址间的相关性减少存储冗余。通过将目标地址分解为区域、页和偏移量 3 部

分,并将区域和页分别替换为区域号和页号,PDeDe 减少了目标地址间的存储冗余。然而,通过编号查询完整地址的过程会引入额外的误预测、延迟和容量开销,导致效率不佳。

BTB-X[8] 和 MBTB[9] 利用空间局部性,分别以页内偏移量和相对偏移量 2 种方式编码目标地址,减少了存储冗余。其中,相对偏移量指目标地址相对分支指令的偏移量,需要使用加法器计算目标地址。相对偏移量则指目标地址在分支指令所处对齐地址边界内的偏移量,虽然在跳转跨越边界时增加了存储开销,但可通过位拼接完成计算。

根据 IPC-1 数据集,图 5 对比了 2 种编码方式下不同位数偏移量可覆盖跳转分支比例。由于编码位数的分布相似,页内偏移量通过免去加法器减少了查询延迟,是一种良好选择。此外,在应用于 ARM(advanced RISC machines)和 Loonarch 64 等指令按 4 字节对齐的架构时,由于合法指令地址低 2 位的值为 0,偏移量不必存储地址的低 2 位。

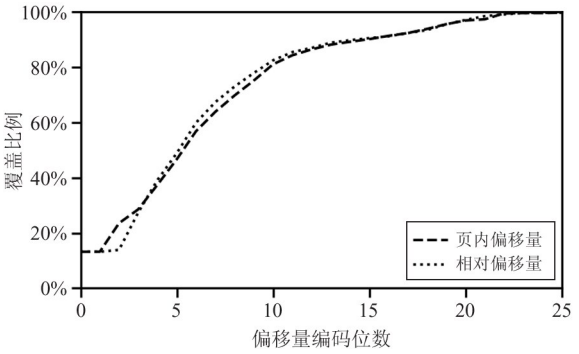

图 5　2 种编码方式下不同位数偏移量可覆盖跳转分支比例

2.2　存储格式设计

为存放编码后长度不一的目标地址,现有设计使用多种 BTB 表项格式进行存储。

多项尺寸 PDeDe[7](PDeDe-multi entry size)和 BTB-X[8] 在不同路存储不同长度的目标偏移量。其中,多项尺寸 PDeDe 以一半的路存储页内跳转。BTB-X 选择根据不同偏移量长度的分支比例为各路的偏移量静态分配存储空间,并将偏移量位数超过 25 的分支存储到 BTB-XC[8] 中,其结构如图 6 所示。为了缓解冲突缺失,BTB-X 根据最近最少使用(LRU)替换算法将新 BTB 项分配到所有偏移量位数足够的路上。例如,偏移量为 3 位的分支可以被分配路 1 ~7 中的任何一路。

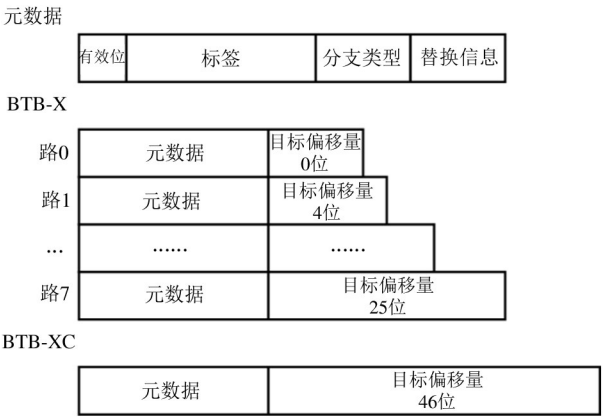

图 6　BTB-X 表项结构

BTB-X 按照动态分支的覆盖比例相近的原则为各路偏移量静态划分了存储空间,存在各个关联组中实际需要存储的热点分支的偏移量位数分布与总体分布可能不符的问题。该问题导致了两方面的不足。一方面,在短偏移量的 BTB 项占用长偏移量的路时,BTB-X 浪费了存储资源。另一方面,由于较长偏移量的 BTB 项共享了较少的路数,BTB-X 存在容易发生冲突的问题。其中,即使总体上分支不浪费地存入各路的概率相等,同一关联组 8 条热点都恰好不浪费地存入各路的概率也仅有不到 6×10^{-8}。

MBTB[9] 通过 2 种表项变种存储不同长度的目标偏移量,实现了对空间分配的动态调整,并通过偏斜哈希[5] 缓解了冲突缺失。根据 MBTB 在正文中的描述,其表项结构如图 7 所示。其中,变种 0 通过完整地址记录 1 个长偏移量分支,而变种 1 可记录 2 个短偏移量分支。MBTB 实际上通过表项表示了一个含有 0 ~2 路的关联组。因此,其偏斜相联的每路存储了一个组相联的关联组。

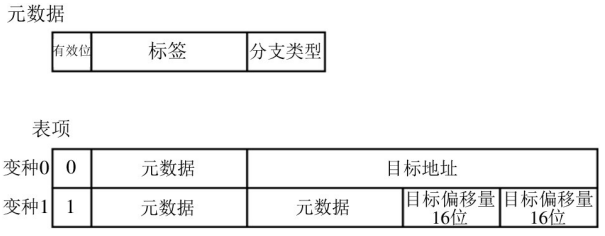

图 7　MBTB 表项结构

MBTB 存在替换算法和存储粒度两方面的不足。在替换算法方面,由于无法利用表项的可重用性信息,其所用的随机替换算法容易导致冲突。而在存储粒度方面,按照若干固定表项格式的存储方式粒度较粗,容易产生空间浪费。

2.3 对现有工作的修正

本文调整了分支类型及标签 2 部分的位数,并修正了 MBTB 标签存在重复信息导致位数过多的问题。

使用间接跳转预测器和 RAS 的实际处理器设计[13]需使用 3 位二进制编码表示 6 种分支类型。而部分基于模拟器的工作[8-9]按照分支类型位数为 2 进行了计算,不适用于带有间接跳转预测器的现代处理器。

本文修正了 MBTB[9]使用标签位数过多的问题,并将修改后的设计记为 MBTB + 。通过如图 8 所示的标签和索引计算方式,MBTB + 可使用与其他设计相同的标签长度。其中,f、g 和 h 为 3 个哈希函数,XOR 表示按位异或,所有路共同使用地址高位哈希值作为标签。其中,若将标签位数减少至 14,相比于使用 28 位标签的 MBTB,MBTB + 在哈希混淆率仍可忽略不计的同时将存储量减至 76% 。

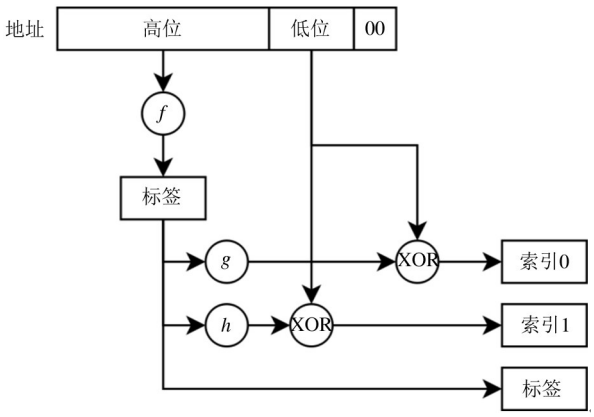

图 8 一种可用于偏斜相联结构的标签和索引计算方式

2.4 对比和观察

表 2 总结并对比了现有压缩 BTB 结构在分配方式、分配粒度、替换算法和偏斜相联 4 方面的特点。其中,动态的分配可以避免实际存储需求与静态划分错配的问题。更细的粒度可以减少多余的偏移量位数。替换算法和偏斜相联结构都是减少冲突的方法,而现有 BTB 未能有效结合这 2 种方法。

表 2 压缩 BTB 结构对比

设计	分配方式	分配粒度	替换算法	偏斜相联
传统 BTB	静态	单一	LRU	无
BTB-X	静态	细	LRU	无
MBTB	动态	粗	随机	有
ZBTB(本文)	动态	细	LRU	有

基于以上观察,本文提出的基于动态 ZBTB,使用了细粒度的动态分配,将偏斜相联结构与替换算法相结合。

3 结构设计

ZBTB 从多角度提升 BTB 的存储效率。其中,3.1 节介绍整体组织方式,3.2 节介绍目标地址的压缩方法,3.3 节介绍元数据的压缩方法,3.4 节介绍替换算法的实现。

3.1 组织方式

ZBTB 对关联组内多路分支信息进行压缩,并在关联组间使用偏斜相联结构缓解冲突。其组织结构及其查询方式如图 9 所示。在进行查询时,ZBTB 使用独立的哈希函数计算索引,并行读取偏斜相联的多个关联组。每个关联组先根据索引从随机存取存储器(random access memory,RAM)中读出压缩数

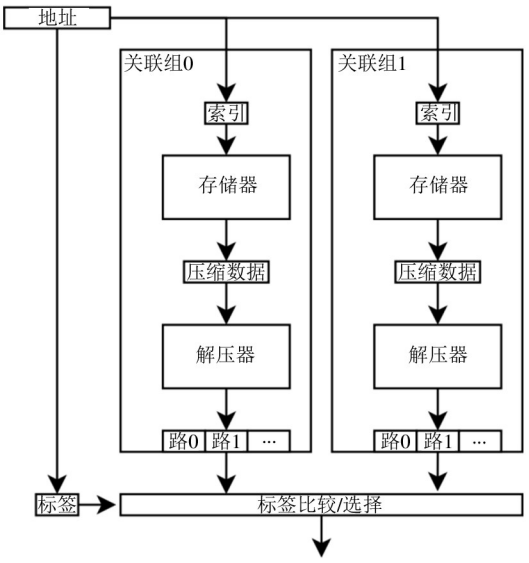

图 9 ZBTB 组织结构及其查询方式

据;再通过解压器得到如多路传统 BTB 表项的解压数据;最终 ZBTB 根据标签比较从各关联组的各路中选出命中分支的类型及其目标地址。相应地,在对关联组进行修改时,ZBTB 先读出并修改解压数据,再将重新压缩所得压缩数据存回 RAM。

压缩数据按需分配存储空间,包括路数、元数据和偏移量 3 段,其结构如图 10 所示。其中,元数据又包括标签、分支类型和偏移量长度码 3 部分。各路元数据和偏移量按照各自长度紧密排放,分别由高位和低位向中间增长,从而共享了存储空间。在偏移量总长度较短时,压缩数据可以存储更多元数据。

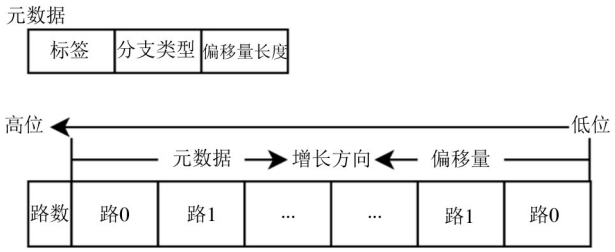

图 10　压缩数据及元数据结构

解压器可根据路数和各路偏移量长度码为各路选出偏移量,并将其与查询地址拼接,从而完成解压。其中,由于包含长度信息的元数据定长且排放紧密,各路元数据被存储于固定位置。解压器可以根据各路的固定位置直接读出长度信息,并行完成各路的解压,从而控制解压过程的延迟。

3.2　目标地址编码

ZBTB 通过页内偏移量的方式编码目标地址,即直接将目标地址的低位用作偏移量。在分支目标地址与其指令地址存在差异的最高位数为 N_B 位的情况下,实际存储的目标偏移量位数 N_Z 不少于 N_B。本文称 N_B 为分支的目标差异位数。

ZBTB 使用可变长编码表示目标偏移量,其编码可表示的偏移量长度构成集合 S_Z。在存储时,ZBTB 从集合 S_Z 中选择为偏移量位数 N_Z,并将其对应的偏移量长度码存储于元数据中。其中,存储空间位数 N_Z 不小于目标差异位数 N_B,有:

$$N_Z = \min_N \{N \mid N \in S_Z,\ N \geqslant N_B\} \tag{1}$$

且偏移量长度码的位数 N_E 有:

$$N_E = \lceil \log_2 \| S_Z \| \rceil \tag{2}$$

给定 S_Z,目标地址占用位数 N_A 为偏移量存储空间位数 N_Z 和偏移量长度码位数 N_E 之和,即:

$$N_A = N_E + N_Z \tag{3}$$

为提高存储效率,ZBTB 可根据分支偏移量位数的概率 $P(N_B)$ 选择恰当的 S_Z,使目标地址占用位数的期望值 $E(N_E)$ 最小。

此外,ZBTB 可用粗粒度的目标地址编码。以位为单位的长度控制虽然能实现更精细的压缩,但是会增加解压器中加法树和选择逻辑的延迟和面积。该设计可选择长度 L_G 作为最小粒度,使得所有偏移量合法长度为 L_G 的整数倍。此时,加法树和选择逻辑可以按照最小粒度组织,从而减小延迟和面积。

3.3　元数据压缩

为了提高元数据的存储效率,在通过哈希压缩标签的基础上,ZBTB 用关联组的路数信息代替了传统 BTB 中每路的有效位,并通过混合编码压缩了分支类型和偏移量长度码。

ZBTB 使用路数代替有效位,节约了有效位的存储开销。由于在摆放时有效路连续,ZBTB 只需要记录当前有效的总路数,无需通过传统有效位独立表示各路是否有效。例如,在最大存储 7 路时,3 位的路数信息至多可以代替 7 位的有效位,从而为每个关联组减少 4 位存储空间。

注意到间接分支无需存储偏移量,ZBTB 将分支类型与偏移量长度码编码为混合编码,进一步提高了元数据的存储效率,其编码方式如图 11 所示。其中,N_E 位的混合编码可表示带有 N_E-2 位偏移量长度码的 3 种直接分支和偏移量长度为 0 的 3 种间

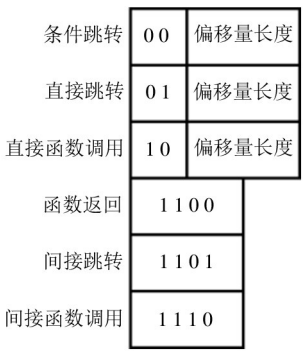

图 11　分支类型及偏移量长度码混合编码方式

接分支,从而一共可表示 $1+2^{N_E-2}$ 种不同的偏移量长度。

3.4 替换算法

ZBTB 在偏斜相联的多个关联组中随机选择,并在关联组内实现替换算法,无需存储额外数据。其中,替换可分为换入和换出 2 步。新表项的换入占据压缩数据的空闲位数,而旧表项的换出则会释放压缩数据的位数。

ZBTB 按照换出顺序存储各路分支信息,将更优先被换出的路存放在编号更大的路上。当压缩信息剩余空间位数小于换入表项位数时,压缩数据会丢弃编号最大的若干路分支信息,从而为换入表项保证存储空间。

通过调整各路分支信息摆放顺序,ZBTB 可以使用 LRU、先入先出(first in first out,FIFO)和随机等替换算法。其中,LRU 和 FIFO 替换算法要求将新分支插入到路 0,随机替换算法要求按均匀分布随机插入。特别地,在实现 LRU 替换算法时,ZBTB 需要将被重用分支重新插入到路 0。

4 实验结果与分析

本节介绍实验所用的基准测试和平台,展示并分析实验结果。

4.1 评估方式

本文用每千条指令误预测(mis-predictions per kilo instructions, MPKI)值衡量 BTB 的性能。具体而言,BTB 的误预测包括缺失和混淆 2 类。其中,缺失代表分支需要跳转但 BTB 未提供分支信息的情况,而混淆则代表 BTB 为查询地址提供了错误的分支信息的情况。设计的 MPKI 值越小,则有效容量越大,从而在实际容量相同时存储效率越高。

本文基于 IPC-1[11] 所发布的数据集进行了实验,通过 5×10^7 条指令进行预热并通过另外 5×10^7 条指令进行评估。该数据集可分为客户端、服务器及 SPEC 等 3 种负载。

4.2 目标地址压缩

为获取实际负载运行过程中表项的统计特性,本文每 1 万条动态指令扫描 1 次 BTB 中存储全部表项。本文模拟了采用 14 位标签的 512 组传统 BTB,对其地址在页内偏移量编码下的位数分布进行了统计。

不同负载中页内偏移量位数的分布有所差异。图 12 展示了 BTB 表项中各偏移量位数出现概率的箱形图及其平均值曲线。其中,14 位以上偏移量出现概率的上四分位数可达下四分位数的 3 倍,存在明显波动。在 BTB-X 中,各路可用偏移量位数按照偏移量分布固定划分,无法按照实际负载偏移量。ZBTB 根据关联组实际使用的偏移量位数动态分配存储空间,能根据负载适应偏移量实际的分布。

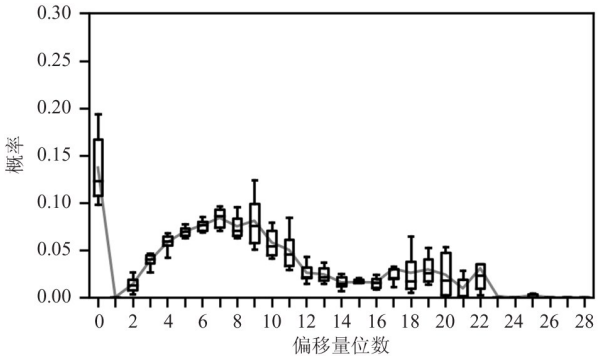

图 12 BTB 表项中各偏移量位数出现概率

通过选择偏移量长度的取值范围,ZBTB 可以降低目标偏移量相关数据的总存储位数期望值。其中,总存储位数期望值为混合编码位数和目标偏移量位数期望值之和。经梯度下降方法优化后,不同长度编码位数下的期望存储位数如表 3 所示。

表 3 不同长度编码位数下的期望存储位数

混合编码位数	目标偏移量位数	总位数
4	10.68	14.68
5	9.27	14.27
6	8.53	14.53

4.3 替换算法

ZBTB 在关联组间使用偏斜相联结构,并在关联组内使用替换算法,通过缓解冲突提高了存储效率。

ZBTB 采用动态分配偏移量和元数据存储空间的策略,每个关联组的路数随实际的空间分配而改

变。图 13 比较了同 RAM 结构达到不同 MPKI 值时的读取位数,即 ZBTB 与传统 BTB 同样使用 13 位标签、1 024 关联组及 LRU 替换算法,且不使用偏斜相联结构。在达到相同的 MPKI 值时,ZBTB 的读出位数相比 2、4 及 6 路的传统 BTB 可分别减少 45%、51%及 54%。其中,仅读出 127 位压缩数据的 ZBTB 可达到与 4 路传统 BTB 相同的 MPKI 值。因此,ZBTB 中平均每路数据约占 32 位存储空间。

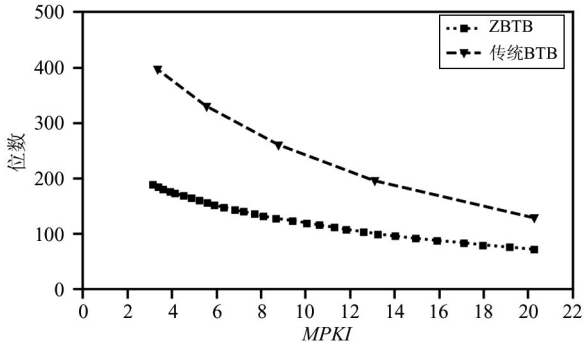

图 13　同 RAM 结构达到不同 MPKI 值时读取位数的比较

ZBTB 通过各路信息的存放顺序表示替换信息,避免了额外存储替换信息导致有效容量降低的问题。其中,替换信息即使只有 1 位,也会使 ZBTB 的平均长度增加 3%,导致存储效率降低。

ZBTB 所选配置为 LRU 替换算法及 2 组偏斜相联结构。在容量为 30 kB 且总压缩数据宽度为 240 的配置下,图 14 对比了不同偏斜相联组数与替换算法的 MPKI 值。其中,在所选配置下,ZBTB 的 MPKI 值可以达到 0.45,同组织结构下相比随机和 FIFO 替换算法降低了 14%。若在同组织结构下取 MPKI 值最低的替换算法,所选配置的MPKI值相比仅使用组相联结构降低了 55%。若继续增加关联组数,由于关联组宽度的进一步缩窄,3 组偏斜相联结构的 MPKI 值相比所选配置反而增加了 23%。

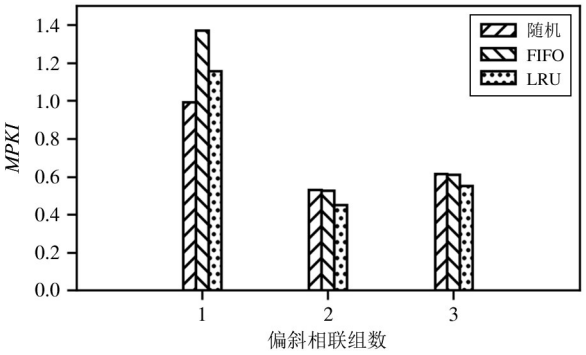

图 14　不同偏斜相联组数与替换算法的 MPKI 值对比

4.4　对 比

本文将 ZBTB 与传统 BTB、BTB-X 和 MBTB + 3 种设计进行了对比。结果表明,ZBTB 在各容量下实现了最高的存储效率。为了保持统一,本文将标签位数设为 13,将分支类型位数设为 3,并将目标地址不变的间接跳转和间接函数调用视为直接分支。

本文所对比偏移量配置采用来自文献[8,9]的典型值。其中,各 BTB 结构的地址或偏移量位数配置如表 4 所示。传统 BTB、BTB-X 和 ZBTB 使用 LRU 替换算法,MBTB + 使用文献[8]所用的随机替换。

表 4　各 BTB 结构的地址或偏移量位数配置

BTB 结构	地址或偏移量位数
传统 BTB	46
BTB-X	0、4、5、7、9、11、19 或 25
MBTB +	16 或 46
ZBTB	0、5、8、11、15、19、22、25 或 46

基于上述配置,本文对比了在不同容量下各 BTB 结构的 MPKI 值,其结果如图 15 所示。与 BTB-X 相比,ZBTB 动态确定各路的目标地址长度并通过偏斜相联结构缓解了冲突,在 15.5 kB、31.0 kB 和 62.0 kB容量下分别将 MPKI 值降低了 44%、86%和 90%。而与 MBTB + 相比,ZBTB 能够更加灵活地为偏移量分配存储空间并通过关联组内的替换算法提高了所存储表项的可重用性,在 16.8 kB、33.5 kB 和 67.0 kB 容量下分别将 MPKI 值降低了 46%、66%

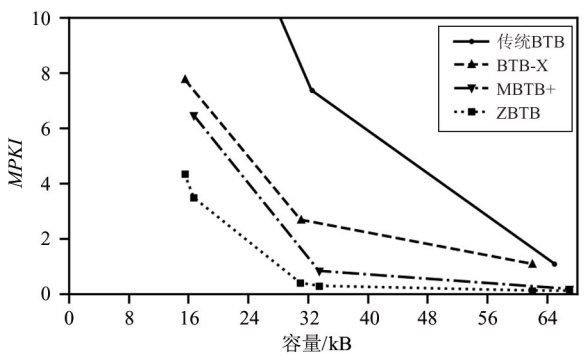

图 15　不同容量下各 BTB 实现的 MPKI 值比较

和40%。

4.5 可实现性分析

在传统结构的基础上,ZBTB 增加了压缩和解压逻辑,其查询和插入存在额外延迟和面积开销,但不影响可实现性。相比于约 32 kB 的存储器,少量 128 位移位器及 7 位加法器的面积可忽略不计。

在查询时,ZBTB 使用加法树并行计算各路偏移量在压缩数据中的起始位数,并通过移位选出命中路的偏移量。相比于传统的标签比较及并行选择逻辑,加法树和偏移量的选择可能导致访问延迟增加约 1 个时钟周期。该延迟远小于指令高速缓存缺失及分支误预测的延迟。特别地,由于每份压缩数据中路 0 和路 1 的偏移量不需要通过加法树获取,可以无需增加访问延迟。

在进行修改时,ZBTB 可通过筒形移位及拼接操作在压缩数据中插入新的项,可使用 1 个额外的时钟周期来完成插入操作。由于正在进行的修改操作也可以被查询获取,查询操作的延迟可以不对性能产生影响。

5 结 论

本文提出了一种基于动态压缩的 BTB——ZBTB。ZBTB 在关联组间通过偏斜相联结构缓解冲突,在关联组内通过页内偏移量编码动态压缩了目标地址,并以替换顺序存储的方式避免了替换信息的存储,从多方面提升存储效率。实验结果表明,在约 16 kB、32 kB 及 64 kB 的容量下,ZBTB 相比现有设计可将 MPKI 值分别降低 44%、66% 及 40%。

参考文献

[1] 胡伟武, 汪文祥, 吴瑞阳, 等. 龙芯指令系统架构技术[J]. 计算机研究与发展, 2023,60(1):2-16.

[2] ISHII Y, LEE J, NATHELLA K, et al. Re-establishing fetch-directed instruction prefetching: an industry perspective[C]//Proceedings of the 2021 IEEE International Symposium on Performance Analysis of Systems and Software. Stony Brook, USA: IEEE, 2022:172-182.

[3] ISHII Y, LEE J, ZHOU W, et al. Rebasing instruction prefetching: an industry perspective[J]. IEEE Computer Architecture Letters, 2020,19(2):147-150.

[4] ADIGA N, BONANNO J, COLLURA A, et al. The IBM z15 high frequency mainframe branch predictor industrial product[C]//Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture. Valencia, Spain: IEEE, 2020:27-39.

[5] SEZNEC A. A case for two-way skewed-associative caches[C]//Proceedings of the 20th International Symposium on Computer Architecture. San Diego, USA: IEEE, 1993: 169-178.

[6] JALEEL A, THEOBALD K B, STEELY S C, et al. High performance cache replacement using re-reference interval prediction (RRIP)[C]// Proceedings of the 37th International Symposium on Computer Architecture. Saint-Malo, France: IEEE, 2010:60-71.

[7] SOUNDARARAJAN N, BRAUN P, KHAN T A, et al. PDeDe: partitioned, deduplicated, delta branch target buffer[C]//Proceedings of the 54th Annual IEEE/ACM International Symposium on Microarchitecture. Virtual, Greece: IEEE, 2021:779-791

[8] ASHEIM T, GROT B, KUMAR R. A storage-effective BTB organization for servers[C]//Proceedings of the 2023 IEEE International Symposium on High-Performance Computer Architecture. Montreal, Canada: IEEE, 2023: 1154-1167.

[9] GUPTA V, PANDA B. Micro BTB: a high performance and storage efficient last-level branch target buffer for servers[C]//Proceedings of the 19th ACM International Conference on Computing Frontiers. Turin, Italy: ACM, 2022:12-20.

[10] KUMAR R, HUANG C C, GROT B, et al. Boomerang: a metadata-free architecture for control flow delivery[C]//Proceedings of the 2017 IEEE International Symposium on High-Performance Computer Architecture. Austin, USA: IEEE, 2017:493-504.

[11] PUGSLEY S. 1st Instruction Prefetching Championship [EB/OL]. [2023-05-16]. https://research.ece.ncsu.edu/ipc/welcome/:Intel.

[12] REINMAN G, AUSTIN T, CALDER B. A scalable front-end architecture for fast instruction delivery[J]. ACM SIGARCH Computer Architecture News, 1999,27(2):234-245.

[13] ZOU J R, TANG D, CAI Y, et al. A design of fetch tar-

get buffer implemented on Xiangshan processor[C]//Proceedings of the International Conference on Cloud Computing, Internet of Things, and Computer Applications. Luoyang, China: IEEE, 2022:1-7.

[14] PERAIS A, SHEIKH R, YEN L, et al. Elastic instruction fetching[C]//Proceedings of the 2019 IEEE International Symposium on High Performance Computer Architecture. Washington, USA: IEEE, 2019:478-490.

[15] YEH T Y, MARR D T, PATT Y N. Increasing the instruction fetch rate via multiple branch prediction and a branch address cache[C]//Proceedings of the 7th ACM International Conference on Supercomputing. Tokyo, Japan: ACM, 1993:67-76.

[16] SEZNEC A. Don't use the page number, but a pointer to it[C]//Proceedings of the 23rd International Symposium on Computer Architecture. Philadelphia, USA: IEEE, 1996:104-113.

A storage efficient last-level branch target buffer based on dynamic compression

TAN Hongze, WANG Jian

(State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190)

(University of Chinese Academy of Sciences, Beijing 100049)

Abstract

With the increasing size and complexity of software systems, the massive instructions bring frequent misses to instruction caches and branch target buffers (BTBs) and hurt central processing unit (CPU) performance. Modern industry CPU designs utilize sufficiently large multi-level BTBs in decoupled front end to reduce performance degradation from misses and consequently result in vast BTB storage requirements. However, current compressed BTBs use statical allocation policies that cannot adapt to upcoming branches. To overcome the limitations of current BTBs, this work proposes a dynamically compressed BTB called zipped branch target buffer (ZBTB). ZBTB uses an adaptive allocation policy enabled by the employment of variable length target offset with a storage-free least-recently-used (LRU) replacement and skewed associativity to reduce conflictions. Evaluate ZBTB on traces from the First Instruction Prefetching Championship (IPC-1). Compared with the state-of-the-art storage-efficient BTBs, ZBTB can reduce the misses by over 66% with the 33.5 kB storage budget.

Key words: branch prediction, branch target buffer (BTB), compression, skewed associativity