基于跳转轨迹的分支目标缓冲研究

熊振亚[1] 林正浩[2] 任浩琪[1]
(同济大学电子与信息工程学院 上海 200092)[1] (同济大学微电子中心 上海 200092)[2]

摘 要 现代计算机体系结构受两个方面的困扰:性能和能耗。为降低嵌入式处理器日益增长的功耗,提出基于跳转轨迹的分支目标缓冲结构(TG-BTB)。与传统分支目标缓冲每次提取指令时需要查询分支目标缓冲不同,TG-BTB只在执行轨迹预测为跳转时才查询分支目标缓冲。该结构通过在程序执行过程中动态分析跳转轨迹行为,可以实现只在轨迹跳转时查询分支目标缓冲,从而降低功耗。在动态分析过程中首先提取记录两条跳转分支指令之间的指令间隔,然后将提取的指令间隔存储在TG-BTB中,最后根据存储在TG-BTB中的指令间隔决定是否需要查询BTB。基于基准测试向量进行模型验证和性能测试,实验结果表明TG-BTB降低了81%的BTB查询能耗。

关键词 跳转轨迹,指令间隔,分支目标缓冲,能耗
中图法分类号 TP301,TP332 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2017.03.042

Efficient BTB Based on Taken Trace

XIONG Zhen-ya[1] LIN Zheng-hao[2] REN Hao-qi[1]
(School of Electronics and Information Engineering, Tongji University, Shanghai 200092, China)[1]
(Microelectronics Center, Tongji University, Shanghai 200092, China)[2]

Abstract Computer architecture is beset with two opposing things: performance and energy consumption. To reduce the increasing energy consumption of embedded processor, we proposed a taken trace branch target buffer (TG-BTB) which is an energy efficient BTB scheme for embedded processors. Unlike the conventional BTB scheme, which requires lookup BTB every instruction fetch, the TG-BTB need lookup BTB only when the trace is a taken trace. This structure dynamically analyzes the trace behavior during program execution, and TG-BTB can achieve lookup BTB per taken trace and reduce the energy consumption of BTB lookup. In the process of dynamic analyzing, TG-BTB detects the instruction interval between two taken instructions firstly, and then stores this value into TG-BTB. Finally, the scheme determines to perform BTB lookup or not according to the instruction interval. The experimental results demonstrate TG-BTB achieves 81% energy consumption reduction compared to the conventional BTB scheme.

Keywords Taken trace, Instruction Interval, BTB, Energy consumption

1 引言

分支指令是一种特殊的指令,可以使程序从正在执行的顺序指令流跳转到程序的其他位置。分支预测通过预测由分支引入的指令流的转移来增加处理器流水线内指令处理的并行性。分支预测通过使处理器在提取分支指令时提前提取指令流中的下一条指令而不是等待分支结果来提高处理器的性能,在当前处理器中是一种猜测性的优化。

分支预测包含预测分支指令的两个部分:1)分支指令跳转的方向;2)分支目标地址。分支指令跳转的方向是指分支跳转的结果,比如分支结果是否跳转至一个新的位置(分支跳转),或者分支指令为不跳转,后续指令依旧跟随之前的指令流。对于分支跳转,分支目标指令就是下一条即将执行的指令。

在分支预测中主要的结构就是分支目标缓冲[1](Branch Target Buffer,BTB)。BTB通过记录特定分支指令所执行的最后一个分支目标作为最后一级预测器。在大多数结构中,BTB就是一个表,将分支指令第一次跳转后的分支源地址和分支目标地址存储其中。在后续执行中再次遇到该分支指令时,处理器通过查找BTB来预测分支是否跳转,并通过读取BTB中存储的分支目标地址来获取分支目标。如果分支目标地址与前一次分支指令的分支目标地址一致,那么BTB的预测就是正确的。

为了提高处理器的性能,分支预测需要保证准确性和实时性并重。一方面,在理想情况下,BTB应该足够大来装下整个工作集以提高BTB的准确性。另一方面,大容量的BTB需要更长的访问延时,使得分支预测时间变长,这样会抵消由于大容量BTB所带来的准确性提高的益处。此外,大容量BTB需要更多的硬件资源开销。由于应用程序的复杂性,很难找到一种适合所有情况的BTB结构来权衡BTB的准确

到稿日期:2016-12-12 返修日期:2017-01-12
熊振亚(1985-),男,博士生,主要研究方向为计算机体系结构、缓存结构设计,E-mail:xiongzhenya@126.com;林正浩(1957-),男,硕士,教授,博士生导师,主要研究方向为高性能处理器设计;任浩琪(1980-),男,博士生,讲师,主要研究方向为微处理器设计及可测性。

性、BTB 的访问延迟以及硬件开销之间的关系。此外，由于 BTB 的结构与采用 SRAM 的缓存结构一致，在传统 BTB 结构中每次提取指令时都需要访问 BTB，因此消耗了大量功耗。在 Pentium Pro 中[2]，512 个表项的 BTB 占整个处理器能耗的 5%左右；在两级 BTB 结构中，BTB 能耗占整个处理器能耗的 7.4%[3]。基于以上分析，BTB 的查询占据了大量的功耗，其中大多数的查询都是多余的。本文提出了一种基于分支跳转轨迹的 BTB(Trace Guide BTB，TG-BTB)，旨在通过最小化查询次数来减少 BTB 的能耗。本方案的中心思想是只有当执行轨迹可能跳转时才查询 BTB。本文开发了一种动态的跳转轨迹分析技术，在程序执行期间收集足够的跳转轨迹信息。根据历史执行的分析数据，在程序执行时可以有条件地避免不必要的 BTB 查询，从而减少能耗。本文的主要工作及创新点如下：

1)所提出的降低 BTB 功耗的方案是一种与软件优化无关的技术，在没有任何编译器工具的情况下，可以在程序执行期间动态地分析所采集的执行轨迹。

2)所提出的 TG-BTB 结构可以实现只有在轨迹点可能跳转时才查询 BTB，这比每次提取指令时都查询 BTB 的传统结构节省了大量能耗。由于一条跳转的轨迹点包含更多的基本指令块，因此所提的结构与最新的 BTB 结构——AirBTB 相比还要节省能耗。

3)通过在 BTB 表项中存储额外的元数据，实现了在查询 BTB 时不用访问标签缓存，从而降低了功耗。

4)通过模拟验证，执行基准测试向量 SPEC2000，TG-BTB 可以将传统 BTB 的查询能耗平均减少 81%。由于 BTB 的能耗占整个处理器能耗的 6.8%[22]，因此本结构可以减少整个处理器 5.5%的能耗。

2 相关工作

预测分支目标是分支目标预测中非常重要的一步，即使分支预测器预测分支指令为跳转，但如果分支目标地址没有存储在 BTB 中，那么下一条指令的提取地址就不能在分支指令执行完成之前获得。在这种情况下，处理器就会暂停前端流水线或者提取下一条指令。一旦分支指令在流水线中执行，如果该分支指令被证实为跳转，那么处理器将会从正确的地址中重新提取指令。因此，理想情况下，BTB 需要足够大的容量来记录程序中所有跳转的分支指令及其分支目标地址。

在此之前已经有很多的研究关注这一领域。现在，很多指令足迹比较大的程序都含有大量的静态分支[4-6]。特别地，大家普遍认为高分支指令预测错误率是由于复杂的分支模式引起的，然而 Annavaram 等[5]却指出高分支预测错误率是由于大量的静态分支引起的。一旦在分支跳转预测中使用的表足够大，BTB 的表项数达到 16k 时，就会显著降低预测错误率[7]。

Sussenguth[8]第一次提出分支目标缓冲(BTB)的概念。Lee 和 Smith[9]提出了更加详细的 BTB 设计，包括层次化的 BTB 设计。Intel Nehalem 处理器及最新的 Intel Core i7 处理器都包含了两级 BTB，虽然其详细结构不得而知，但是它是目前处理器中最关键的部件[10]。层次化的 BTB 致力于达到大容量 BTB 的准确度。在层次化 BTB 中，关键是要避免大容量 BTB 造成的面积开销以及潜在的较长的访问时间。一种方法是通过对其信息进行编码来降低 BTB 的面积开销[11-13]。在某些情况下，将 BTB 分成几个不同的功能部件来达到减少面积开销的目的，但是这会增加访问时间。

除了使用 RAS 来处理返回指令外[14]，BTB 对提高间接分支预测准确率并无实质的作用。提高间接分支指令预测准确率大多通过基于访问模式或者基于历史的预测来实现[15]。很多研究中提出使用大容量但访问速度长的分支预测器。Seznec 等[16]提出了一种大容量、访问时间慢但是准确度高的分支预测器来代替小容量、速度快但是准确不高的预测器。王国澎等[17]采用索引散列算法，通过散列表来实现对 BTB 的快速访问。

Petrov 和 Orailoglu[18]提出了一种应用程序可定制分支目标缓冲器(ACBTB)。ACBTB 是一种依赖编译器的技术，通过使用应用程序的精确控制流信息，只有在执行分支指令时才访问 BTB。由于控制流信息必须在编译和链接阶段提取出来，因此这是一种静态的方法，不适用于现有的可执行程序。还有一种方法是预译码，通过在指令提取阶段探测该指令是否为分支指令，但是这种方法的缺点是预译码位只有在取指段的最后才可用，这会导致严重的性能损失。Parikh 等[19]提出一种基于硬件的分支预测探测器(PPD)，通过编译器的提示和预译码位提前探测出是否可以避免访问分支预测器和 BTB。由于要求 PPD 自身的查找早于分支预测器和 BTB，因此会增加流水线的长度。此外，不管 BTB 是否访问，PPD 都会消耗额外的功耗。Kaynak 等[21]提出了 AirBTB，其组织结构与 L1 缓存一一对应，每个 BTB 表项存储 3 个分支目标地址，分别对应缓存行中的分支指令。因此，只有从 L1 缓存中提取指令行时才访问 AirBTB，可以降低功耗。但是，由于 AirBTB 中存储的数据量是传统 BTB 结构的 3.5 倍，且大容量的 BTB 自身的静态功耗和每次访问的动态功耗都会增加，这会抵消掉因减少 AirBTB 访问而减少的功耗。

3 轨迹跳转 BTB

3.1 控制流信息

BTB 的目的在于在指令进入处理器前端流水线之前尽早提供分支识别以及提供分支指令的类型，这就需要关于控制流结构的相关知识以便于有效地访问 BTB。程序的控制结构通常由控制流程图(CFG)表示。流程图中的节点对应一段基本的代码，而边沿则代表在执行过程中相应基本代码块之间的跳转。一个基本块是具有单个入口和出口的连续指令，因此诸如分支指令的跳转指令只能是基本块的最后一条指令。

图 1(a)是一个包含两个条件语句的循环程序，该段代码的 CFG 表示见图 1(b)。基本块 *B2* 和 *B3* 对应第一条件语句中的“then”和“else”子句，而基本块 *B5* 对应第二条件语句中的“then”部分。图 1(c)是给定循环的可能流程图。两条分支指令分支 1 和分支 2 分别在块 *B1* 和块 *B4* 的结束处决定是否跳转。基本块 *B2* 以直接无条件跳转结束，其目的是直接跳转到第一条件语句的结合点。如图 1(c)中所示，在指令执

行之后就能知道两条执行的分支指令之间指令数目的距离。例如，在分支指令 1 跳转的情况下，下一条分支指令是在执行 |B3|+|B4| 条指令之后才执行分支指令 2，其中 |Bn| 表示一个基本块中指令的数目。在分支指令 1 不跳转的情况下，下一条分支指令是在 |B2| 条指令之后。显然，在分支指令 1 执行后有两条不同的执行路径。一条路径是通过基本块 B3 和 B4 到达分支指令 2，另一条路径是通过基本块 B2 和 B4 到达分支指令 2。因此，通过利用包含 CFG 的程序信息以及跳转指令之间所有可能执行路径的指令数，并简单地对在分支跳转指令之后执行的指令进行计数来有效跟踪跳转指令的路径，然后根据分支指令的起始方向，后续的分支指令可以准确地被定位。对于这种体系结构，不需要每个时钟周期都查找 BTB；相反，只需要在计数逻辑表明该条指令为分支指令时查询 BTB 即可。

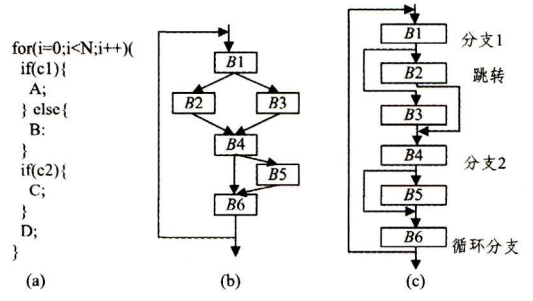

图 1　控制流结构示意图

与基本指令块相对应，本文将两个连续跳转的分支指令之间的指令序列定义为指令跳转轨迹。如图 2 所示，阴影部分是分支跳转的执行轨迹，对于分支指令 1，其跳转轨迹为 B3 和 B4；由于指令块 B6 的最后一条指令为循环分支，对于分支指令 2，其跳转轨迹只包含指令块 B6。如果分支指令 2 不跳转，那么分支指令 1 后的跳转轨迹包含 B3－B6 共 4 个指令块。显然，分支跳转执行轨迹的长度大于或等于一个基本块的长度。本结构的目的不是仅仅在一个基本块内不需要访问 BTB，而是只在分支指令可能跳转时才访问 BTB。

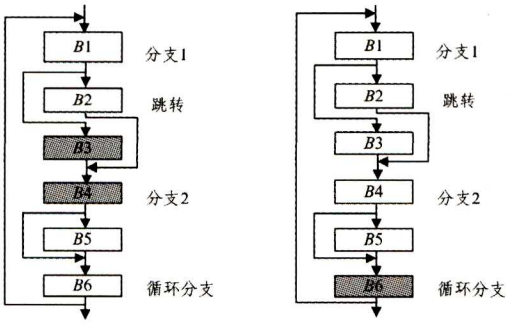

图 2　跳转轨迹示意图

3.2　TG-BTB 的表项构成

BTB 的作用是早于处理核识别分支类型并提供正确的分支目标地址。这种早期分支识别对于流水线处理器特别是在固定的流水线段产生分支判断和分支目标地址的处理器而言是非常重要的。因此，需要在 BTB 的每个表项中存储足够多的信息，以保证整体的性能。图 3 是 TG-BTB 的每个表项的构成。

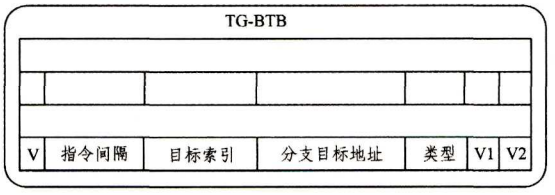

图 3　TG-BTB 每个表项包含的元数据

TG-BTB 每个表项中包含 1 个指令间隔域、1 个目标索引域、1 个分支目标地址以及分支指令类型。指令间隔域代表该跳转分支指令到下一条跳转分支指令之间间隔的指令数目；目标索引域用于直接索引 TG-BTB，它包含分支目标地址对应于 TG-BTB 的索引号以及所在的路组号，通过目标索引域直接找到分支目标在 TG-BTB 中的表项，不需要进行标签比较。与传统 BTB 结构相比，其可以节省大量能耗。由于 BTB 的表项与指令缓存中的指令行不是一一对应的，两个独立模块之间的存储关系没有必然联系，因此对于跳转的分支指令或者直接跳转指令，需要分支目标地址，以便使用该地址从 L1 指令缓存中取出正确的指令行。与传统 BTB 结构一致，分支目标地址域用于提供分支目标地址。类型域表明分支指令的类型，比如条件分支指令、跳转、函数调用、间接分支指令等。类型域还可以用于额外提供有利于分支预测的分支类型信息。最后，TG-BTB 的每个表项还包含 3 个不同的有效位。

在 TG-BTB 结构中，只有指令间隔域与程序执行行为有关，其他域只与 TG-BTB 的表项数有关。对于固定的表项数，目标索引域和分支目标地址域的大小都是恒定的，因此需要确定指令间隔域的大小。指令间隔域的容量应该足够大，以记录大多数跳转轨迹。诚然，指令间隔合适的长度取决于应用程序的动态行为。图 4 给出了 SPEC2000 中跳转指令间隔分布图。从图中可以看到，98% 的跳转指令间隔都小于 64 条指令，在能耗、硬件开销和性能之间的最佳折衷是将跳转轨迹长度设为 64，因此可以将指令间隔域的大小设为 6 位。

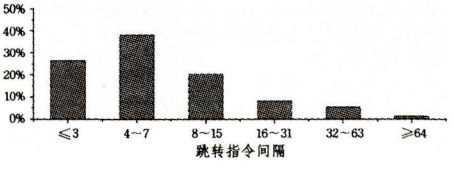

图 4　跳转指令间隔分布

在 TG-BTB 的每个表项中增加共 V，V1 和 V2 共 3 个有效位，其中 V 代表整个表项是否有效，V1 代表是否有其他轨迹跳入该表项，V2 代表分支目标索引域是否有效。有效位主要用于在替换时保持存储在 TG-BTB 中的轨迹的完整性。当 TG-BTB 发生替换时，首先根据有效位 V 判断该表项是否有效。如果 V 无效，直接将新的跳转分支轨迹填充进 TG-BTB 中，然后将该表项的索引号（包括索引位拼接上路组号）加载进临时寄存器，执行后续操作。如果 V 有效，再选择 V1 或/和 V2 无效的路组。因为此时 V1 无效，表明没有其他分支跳转轨迹跳入该表项，而 V2 无效表明该分支指令对应的目标轨迹没有存储在 TG-BTB 中。所以替换 V1 和 V2 无效的表项可以最大程度地保持 TG-BTB 中轨迹的连续性和完整性。如果替换时 V1 有效，则需要将该表项中分支源索引

域指向的 TG-BTB 中对应表项中的 V2 设为无效，以确保执行到分支源对应的跳转轨迹上时引用错误。如果该表项中的 V2 位有效，在替换该表项时需要将分支目标索引指向的 TG-BTB 表项的 V1 设为无效，以表明该表项没有其他分支跳转轨迹跳入该表项。如果 V，V1 和 V2 这 3 个有效位都有效，则按照 LRU 替换策略替换 TG-BTB 中的一个表项。这样就可以确保存储在 TG-BTB 中的分支轨迹是最有效的分支轨迹，从而最大程度地减少对 BTB 的访问。

3.3 整体结构

TG-BTB 的查询方式依赖于之前执行的轨迹有条件地避免 BTB 查询，关键是在执行期间如何分析并获取所执行指令的跳转轨迹。与基于编译器提取执行轨迹的 ACBTB 中提出的方法不同[18]，本文提出的 TG-BTB 不依赖于任何的软件系统(包括编译器)，是一种硬件结构，整个系统结构如图 5 所示。

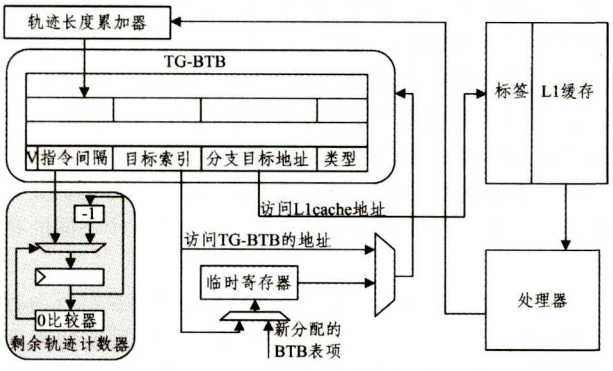

图 5 使用 TG-BTB 的整体结构图

与传统 BTB 结构相比，该系统增加了 3 个模块：剩余轨迹计数器、轨迹长度累加器以及临时寄存器。剩余轨迹计数器用于表明当前指令是否位于一条跳转轨迹中，其初始值为“0”。当 BTB 命中时，剩余轨迹计数器被设置为从 BTB 中命中的表项中读取的跳转轨迹长度值。每提取一条指令，剩余轨迹计数器的值就减“1”，在剩余轨迹计数器的值为“0”之前，都表明此时提取的指令都位于一条跳转轨迹内，并且没有遇到可能跳转的分支指令，因此可以避免访问 BTB 以节省能耗。对于另一种情况，当剩余轨迹计数器的值为“0”时，这意味着当前提取的指令可能是一条跳转的分支指令。此时，就必须查找分支预测器和 BTB 以确定分支预测是否跳转以及分支目标地址。

轨迹长度累加器用于计算两条跳转分支指令之间的指令间隔数，其主要结构为一个 6 位的寄存器。为了能将轨迹长度累加器的值写回正确的分支目标地址对应的 BTB 表项的相应域中，需要在程序执行过程中增加一个临时寄存器来存储上一次跳转分支指令的索引位和路组号，其初始值为“0”。有两种情况需要设置这个临时寄存器：1)在给新的跳转分支指令分配 BTB 表项时，需要将临时寄存器的值设为新分配的 BTB 表项号；2)如果 BTB 命中且其预测正确，需要将临时寄存器的值设为命中的 BTB 的表项号。

3.4 跳转轨迹动态分析

图 6 示出了 TG-BTB 查询结构中动态分支跳转轨迹的情况。BTB 的查询在指令提取阶段进行，而实际的分支结果(包括是否跳转以及分支目标地址等)需要等到指令执行段(EX)才能确定。从流水线中清除由于错误预测而误提取的指令也是在 EX 段执行。从图中可以看到，为了简单说明，将整个动态分析过程分成 7 种可能的情况。这些情况的特性包括由错误预测带来的周期代价总结如表 1 所列，每种情况的详细说明如下。

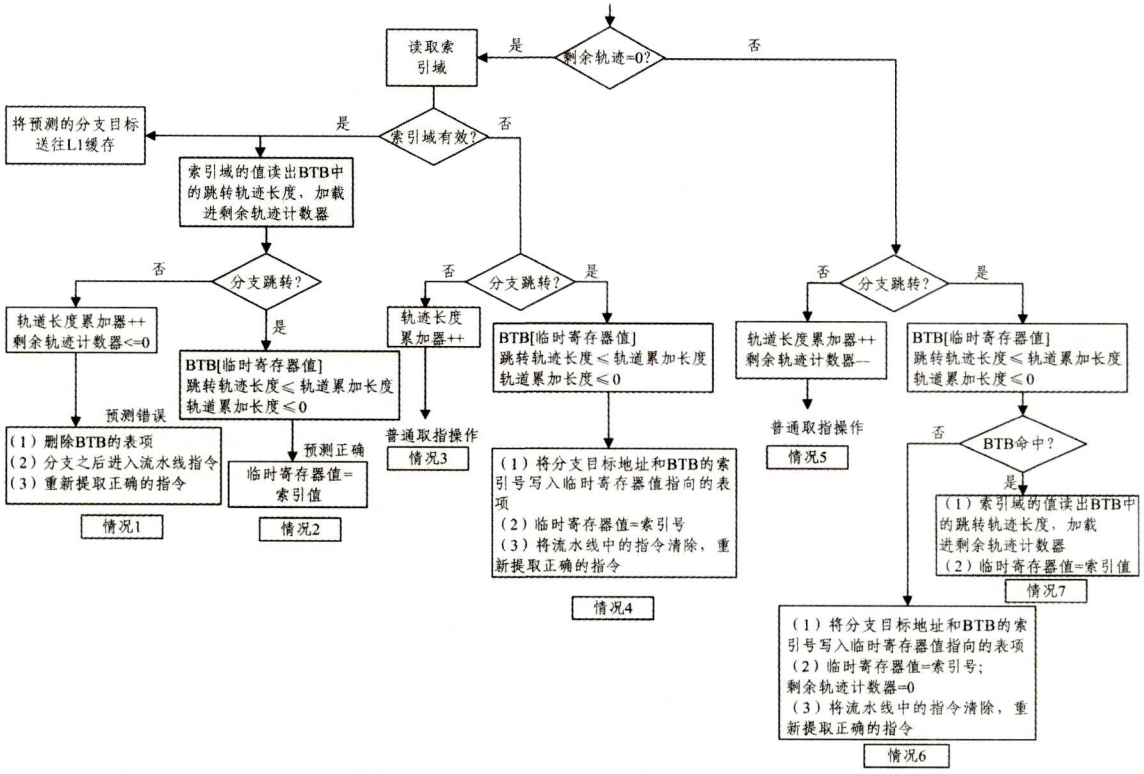

图 6 TG-BTB 查询结构中动态跳转轨迹分析

表 1　TG-BTB 中 7 条可能的访问情况的代价

情况	BTB 查找	命中/缺失	预测	结果	BTB 在 E 段查找	代价
1	是	命中	跳转	不跳转	—	2
2	是	命中	跳转	跳转	—	0
3	是	缺失	—	不跳转	—	0
4	是	缺失	—	跳转	—	2
5	—	—	—	不跳转	—	0
6	—	—	—	跳转	是/命中	3/4
7	—	—	—	跳转	是/缺失	1/2

情况 1　在此情况中，因为指令在 BTB 中命中并且被预测为分支跳转，所以可以从命中的 BTB 表项中检索到相应的跳转轨迹长度，并且在指令取指段将剩余轨迹计数器的值设为从 BTB 相应表项中读取的跳转轨迹长度值。接下来，在指令执行段(EX)中分支指令被执行，得出真实结果为不跳转，这是一次错误预测。此时，剩余轨迹计数器的值必须重置为“0”，并且轨迹长度累加器继续累加跳转轨迹的长度。对于 ARM 指令集，分支判断在指令译码(ID)段产生。需要注意的是，ID 段将与流水线中的 IF 段重叠。为了避免硬件冲突，剩余轨迹计数器只在 IF 段的第二阶段以及 ID 段的第一阶段中改变。在指令执行段，由于分支预测错误，需要将流水线中分支指令之后的指令清除，删除 BTB 的表项，重新从正确的指令流方向提取指令，这会带来 2 个时钟周期的代价。

情况 2　与情况 1 中分支预测错误的情况不同，在该情况中预测器的预测结果是正确的。在 IF 段将剩余轨迹计数器设为检索到的跳转轨迹长度值，然后将轨迹长度累加器恢复到由临时寄存器索引的先前跳转分支指令对应的表项，将轨迹长度累加器重置为“0”，以累加下一条跳转轨迹长度。最后，临时寄存器的值在 EX 段被置为命中的 BTB 的表项号。由于分支预测正确，因此没有任何损失。

情况 3　该情况是执行非分支指令。因此，只需要每执行一条指令就将轨迹长度累加器加“1”，以准确地记录每条跳转轨迹的长度。显然，执行该情况没有任何的损失。

情况 4　由于 BTB 缺失，该取指指令被认为是一条非分支指令或者是一条不跳转的分支指令，但是在 EX 段被确定为是一条跳转的分支指令。因此，轨迹长度累加器的值需要写回由临时寄存器中存储的 BTB 表项值所指向的 BTB 表项的跳转轨迹长度域中，然后将轨迹长度累加器的值重置为“0”，以累加下一条跳转轨迹长度。接着在 EX 段中给该跳转分支指令分配一个 BTB 表项。最后，需要将之前提取的指令从流水线中清除，重新从正确的路径中取指令。执行该情况的代价为 2 个时钟周期。

情况 5　与情况 3 类似，该情况也是执行非分支指令。唯一的不同点是由于剩余轨迹计数器不等“0”，因此在该情况中不用查询 BTB。需要注意的是，除了将轨迹长度累加器加“1”外，剩余轨迹计数器需要在每次取指令时都减“1”。

情况 6　由于剩余轨迹计数器的值不等于“0”，因此在 IF 段不用查找 BTB，但是，在 EX 段产生的分支判断为跳转(Taken)。首先，需要将轨迹长度累加器的值写回由临时寄存器中存储的 BTB 表项值所指向的 BTB 表项的跳转轨迹长度域中，然后将轨迹长度累加器的值重置为“0”以累加下一条跳转轨迹长度。接着在给该跳转分支分配 BTB 表项之前，需要检查该分支是否已经存储在 BTB 中。在情况 6 中，由于该跳转分支未存储在 BTB 中，需要给该分支分配一个 BTB 的表项，方法与情况 4 中的一致。必须注意的是，需要将剩余轨迹计数器的值置为“0”。由于在 EX 段查询 BTB 已不可避免，在最坏情况下会与在 IF 段查询 BTB 的操作出现叠加，因此正常情况下的代价为 3 个时钟周期，而最坏情况则为 4 个时钟周期。

情况 7　与情况 6 基本一致，唯一的区别是跳转的分支目标地址已经存储在 BTB 中，故不需要新分配一个 BTB 表项，直接将命中表项中的跳转轨迹长度值作为剩余轨迹计数器的值。对于正常情况，其分支代价为 1 个时钟周期；对于最坏情况，其分支代价为 2 个时钟周期。

现在将以上 7 条情况的总结如下：

1)在情况 1—情况 4 中，剩余轨迹计数器的值为“0”，与传统 BTB 查找结构一致，需要查找 TG-BTB。相反，在情况 5—情况 7 中，由于剩余轨迹计数器的值不为“0”，可以避免查找 TG-BTB。

2)与传统的 BTB 查找方式相比，节省功耗最显著的是情况 5。这是由于在本结构中存储了足够多的信息，使得在程序执行过程中能有条件地跳过 BTB 的查找。

3)本文提出的 TG-BTB 可以实现只有当进入跳转的轨迹时才查找 BTB，这比基于基本指令块查找 BTB 的 ACBTB[18] 和最近提出的 AirBTB[21] 都更加节省能耗。

3.5　TG-BTB 写入和替换

TG-BTB 的写入与替换直接关系到 BTB 的性能，BTB 替换策略的一个主要目标就是使得 BTB 性能达到最高而不消耗额外过多的元数据。本文提出数学模型来描述存储在 BTB 中的元数据对其性能的影响，如式(1)所示：

$$\mathrm{Max}\{\sum_i h(i)[t(i)\,ptt(i)V(i)-(1-t(i))\,ptnt(i)W(i)]\} \tag{1}$$

其中，$h(i)$是执行分支 i 的概率；$t(i)$是分支 i 跳转的概率；$ptt(i)$是分支 i 预测为跳转且实际也是跳转的概率；$ptnt(i)$是分支 i 预测为跳转但实际并未跳转的概率；$V(i)$是分支 i 正确预测节省的周期数；$W(i)$是分支 i 预测错误的时钟周期代价。式中第一个乘积表示分支预测正确而节省的周期数，式中第二个乘积表示分支预测错误而增加的时钟周期数，这两个乘积的差就是每条分支由于使用 BTB 而节省的周期数。由于 BTB 对于预测正确的不跳转分支指令的性能不会有任何改变，因此式中没有包含这条信息。这个简单的式子包含了两点非常重要的信息：

1)如果一条分支指令对性能的提高没有可能，那就不应该填充进 BTB。式(1)已经表明，BTB 应该填充最有可能改善性能的分支指令。与传统的 BTB 结构不同，所提结构采用最有价值的分支指令替代 BTB 中没有足够价值的分支指令，以期获得最大的性能提升。

2)当需要从 BTB 中替换分支时，应该替换对整体性能影

响最小的分支轨迹。基于此概念，替换策略应该替换那些最不可能被引用且最不可能跳转的分支表项。BTB 不会改善任何不跳转分支的性能，因此替换不跳转的分支几乎不会有任何的性能损失。

对于 TG-BTB 而言，通过临时寄存器记录跳转分支的索引号，以确保跳转分支的所有信息可以在下一条分支指令跳转时写入 TG-BTB 中，这就可以确保只有跳转的分支指令才写入 BTB 的表项中，并且可以保证写入位置的正确性，不会带来性能的损失。当新的分支轨迹填充进 TG-BTB 时必然会导致替换的发生。根据前面讨论的理论，替换是对性能提升影响最小的表项。根据缓存替换的经验，最近最少使用替换算法(LRU)是对于缓存系统而言最好的替换策略，诸如主存储器页面替换和缓存行替换[20]。BTB 的表项仅在成功预测分支跳转且为真实跳转时有用，那些很少执行的分支的 BTB 表项是无用的。LRU 替换策略只能表明哪些分支可能被执行，当分支指令执行时其成功预测的概率可以从分支指令的执行历史中估计。TG-BTB 的替换策略就是替换执行概率和分支跳转概率都最小的表项。

对于最坏情况，即多条分支轨迹跳转到同一条分支轨迹的情况，由于在 TG-BTB 的表项中没有对应的域准确表示哪些条轨迹跳转进入该轨迹，因此在出现替换时无法准确地将分支源轨迹的分支目标索引有效位 V2 设为无效。但是这种情况并不影响性能，现在以两条轨迹跳转进入同一条轨迹为例进行详细说明。假设轨迹 01 和轨迹 13 都跳进轨迹 27。当轨迹 27 根据上述替换策略需要被替换出 TG-BTB 时，由于没有存储足够的信息来表明是轨迹 01 还是轨迹 13 跳进轨迹 27，因此无法将轨迹 01 和轨迹 13 的 V2 有效位设为无效，只能保持轨迹 01 和轨迹 13 的 V2 位有效。当轨迹 01 或者轨迹 13 再次被执行时，其轨道中对应的目标索引域表示该轨迹的分支轨迹存储在 TG-BTB 中，直接用目标索引域访问 TG-BTB 的轨迹 27。由于轨迹 27 已经被替换，已经不再是轨迹 01 和轨迹 13 的目标轨迹，因此从中读出的跳转轨迹长度为错误值。虽然跳转轨迹长度值为错误值，但是轨迹 01 或者轨迹 13 中分支目标地址是正确的，可以直接从 L1 指令缓存中提取指令，因此只相当于一次 BTB 缺失，而不会影响整体性能。此时，只需要重新记录正确的跳转轨迹长度，并将其填充进 TG-BTB 中即可建立一条新的跳转轨迹。

4 仿真结果

本文采用 SimpleScalar 仿真器来模拟经典单发射顺序执行的流水线处理器结构，该结构是经典的 5 段流水线结构，并在此基础上搭建出 TG-BTB 的模型。在本仿真中，基准处理器为每次缓存访问提取一条指令，对于 AirBTB 和 TG-BTB，都是每次访问缓存提取一整个缓存行。处理器的主要参数和代价详细地列在表 2 中。SimpleScalar 模拟类 MIPS/PISA 指令集结构的微处理进行仿真，基准测试程序采用 SPEC2000 中的整型测试向量，每个测试向量采用测试输入(test input)。

表 2 处理器基准参数和代价

处理器基本配置	
指令发射宽度	1 条指令/周期
功能单元	1 Int/FP ALU,1 Int/FPMult/Div
L1 指令缓存	16kB,4-way,32B/Block
L1 数据缓存	16kB,4-way,32B/Block
I/D TLB	32-Entry
BTB	1k-Entry
RAS	16-Entry
L2 共享缓存	2MB,32-way,32B/Block
处理器代价	
L1 命中时间	1 时钟周期
分支错误代价	2 时钟周期
L2 命中时间	10 时钟周期
主存访问时间	100 时钟周期

对于功耗评估，本文采用 CACTI6.5，32nm 技术分别对传统 BTB，AirBTB 和 TG-BTB 每次查询的功耗进行仿真，再根据 SimpleScalar 仿真得出的访问次数可以计算出 3 种不同结构的 BTB 查询功耗。

4.1 情况分布

根据表 1 及图 6 关于每种不同情况的操作及代价发现，情况 1—情况 4 在 IF 段需要访问 BTB，而情况 6—情况 7 在 IF 段不需要访问 BTB，但由于分支预测错误会发生跳转，因此在 EX 段还是需要再访问 BTB。为了区分情况分布从而更好地分析节省能耗程度，将 7 种情况分成 3 组，第一组包含情况 1—情况 4，第二组包含情况 5，第三组包含情况 6 和情况 7。表 3 列出了 3 组情况在不同基准测试下的分布情况。

表 3 SPEC2000 下 TG-BTB 执行情况分布/%

SPEC2000	情况 1—情况 4	情况 5	情况 6—情况 7
gcc	37.83	59.24	2.93
gzip	14.26	85.05	0.69
go	17.73	82.04	0.23
vortex	33.83	63.92	2.25
mcf	13.76	85.78	0.46
vpr	18.82	80.75	0.435
parser	21.36	77.48	1.16
art	11.26	88.52	0.22
average	21.11	77.85	1.04

除了 gcc，vortex 和 parser 3 个基准测试向量以外，其余的测试向量的动态行为属于情况 5 的比例超过 80%，这是由于 gcc，vortex 和 parser 3 个基准测试向量在 1k 个表项的 TG-BTB 中的缺失率比较高以及情况 1—情况 4 所占比例高。在 TG-BTB 结构中，能耗的节省主要来自情况 5，因此情况 5 所占比例高是本文想要达到的目标。相反，无论是情况 1—情况 4 还是情况 6—情况 7，都不可避免地要访问 BTB，因此这两者之和所占的比例越小表明节省的能耗会越多。平均来看，情况 1—情况 4 所占比例大约为 21.11%，情况 6—情况 7 所占比例大约为1.04%，而情况 5 所占比例大约为 77.85%。这表明，在不考虑每个表项容量大小的情况下，TG-BTB 比传统的 BTB 可以节省 77.85%的查询能耗。

4.2 能耗比较

通过 CACTI6.5 可以获得传统 BTB[1]，AirBTB[21] 和 TG-BTB 3 种不同结构每次查询 BTB 的能耗，如表 4 所列。从表 4 中可以看出，随着存储容量的增加，每次查询的功耗也相应增加。需要注意的是，对于 TG-BTB，虽然其总容量为

9.25kB，大于传统 BTB 的容量，但是由于在 TG-BTB 中存储了目标索引域，利用目标索引域可以直接指向目标轨迹所在的路组而不需要经过标签比较，因此每次 TG-BTB 的读取访问的能耗相当于访问一个 256 表项直接映射的缓存能耗。这也是 TG-BTB 每次查询能耗小于传统 BTB 的原因。

表 4　3 种不同结构的 BTB 查询能耗/每次

表项数/1K	BTB 标签/nJ	BTB 缓存/nJ	总能耗/nJ	总容量/kB
传统 BTB	0.00136505	0.00163812	0.00300317	6.75
AirBTB	0.00136505	0.00390682	0.00527187	15.75
TG-BTB	0.00136505	0.00096835	0.00233340	9.25

除了 BTB 的表项消耗能量外，剩余长度计数器、跳转轨迹累加器和临时寄存器也会消耗一定的功耗。这 3 个模块相当于 3 个 6 位的寄存器，其功耗大约为 0.00003517nJ。由于临时寄存器的更新只在情况 4，6，7 中发生，从表 3 中可以看出这 3 种情况所占的比例比较低，因此临时寄存器翻转的频率较低，可以忽略临时寄存器的能耗。相反，对于剩余长度计数器和跳转轨迹累加器而言，其在每种情况下都会计数，所以必须考虑这两个模块的功耗。因此，在计算 TG-BTB 能耗时采用式(2)：

$$E_{TB\text{-}BTB}=0.00233340\times 查询次数_{TB\text{-}BTB}+0.00003517\times 取指令次数 \quad (2)$$

将所有 BTB 结构中消耗的能量总和归一化到传统 BTB 结构中，对于每个基准测试向量，传统 BTB 的能耗都是 100%。能耗归一化后所占比例越小，表明其能耗越小，大于 100%表明能耗高于传统的 BTB 结构，而小于 100%表明其能耗低于传统 BTB 结构。图 7 示出了 3 种 BTB 结构的能耗比较。除了 gcc 和 vortex 两个测试向量，AirBTB 可以节省 26.4%～61.5%的 BTB 查询功耗。由于 gcc 和 vortex 中分支比例比较高，再加上 AirBTB 每次查询的能耗是传统 BTB 的1.73倍，因此其能耗比传统 BTB 结构的能耗大。通过过滤掉大多数冗余的 BTB 查询，TG-BTB 可以将整个 BTB 查询能耗降低 67.5%～89.22%，与传统 BTB 结构相比，降低了 81%的 BTB 查询能耗；与 AirBTB 相比，降低了 48.7%的 BTB 查询能耗。因为 BTB 的能耗大约占整个处理器能耗的 6.8%[22]，所以 TG-BTB 平均可以降低整个处理器 81% * 6.8%=5.51%的能耗。

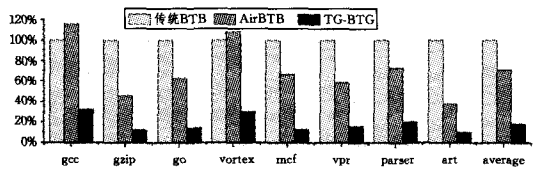

图 7　3 种不同 BTB 结构的查询功耗比较

4.3　性能比较

本文采用每条指令执行的周期数(Cycle Per Instruction，CPI)来评估传统 BTB，AirBTB 和 TG-BTB 的性能。考虑到执行指令的数量和处理器执行时间是常量，CPI 是能直接衡量系统性能的参数。CPI 的值越低表明执行每条指令所需要的时间越少，即性能也越好。对于单发射处理器，CPI 的理想值为“1”。与传统 BTB 结构相比，情况 6—情况 7 会导致额外的损失代价，因此将情况 6—情况 7 称为不利情况。

如果大多数执行情况都按照不利情况执行，CPI 将明显劣化。幸运的是，不利情况发生的概率极低。但根据表 3 所列，执行 SPEC2000 向量时其发生概率为 1.04%。图 8 出示了所有基准测试的 CPI 值。与传统 BTB 相比，TG-BTB 增加了大约 0.76%的 CPI。但 TG-BTB 比传统 BTB 减少了 80%的 BTB 查询功耗，因此是更好的折中。虽然 AirBTB 能使 CPI 降低大约 3%，但是在相同表项数目的情况下，AirBTB 消耗的能耗是 TG-BTB 的 3.2 倍。因此，与传统 BTB 结构和 AirBTB 结构相比，TG-BTB 是一种能效比最高的结构。

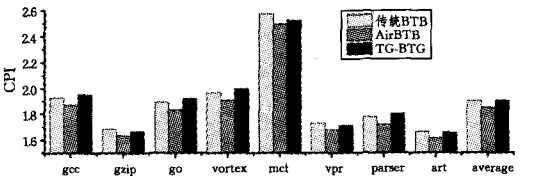

图 8　3 种不同 BTB 结构的性能比较

4.4　面积开销

在传统 BTB 结构中，每个表项的容量大约为 56bit；TG-BTB 每个表项的容量为 75bit；而 AirBTB 每个表项的容量为 130bit。此外，TG-BTB 还包含该 3 个 6 位计数器，这些硬件开销对评估各种 BTB 结构的性价比至关重要。图 9 是 3 种 BTB 结构在不同的表项数下通过 CACTI 仿真得到的面积开销，所有结果都归一化为传统 BTB 结构。从图中可以看出，TG-BTB 的面积开销大约是传统 BTB 面积开销的 1.2 倍，而 AirBTB 的面积开销大约是传统 BTB 的 1.85 倍，是 TG-BTB 面积开销的 1.5 倍。TG-BTB 在 1k 个表项的情况下增加的总容量小于 3kB，远远小于 L1 缓存的容量。因此，通过这点硬件开销的增加而获得的大量查询功耗的降低是一种很好的折中。

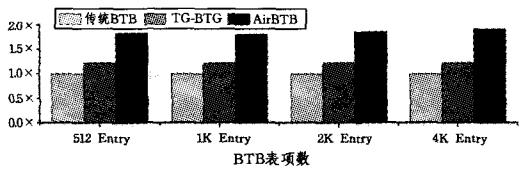

图 9　3 种 BTB 结构的面积开销

结束语　分支指令是一种特殊的指令，可以使得程序改变原有的执行方向，会造成流水线停顿和缓存缺失。在现代处理器中，通常使用分支目标预测器和分支目标缓存来降低分支指令所引起的性能降低。在现有处理器结构中，每次提取指令时都需要访问 BTB，从而造成巨大的能耗开销。本文提出基于动态跳转轨迹的 BTB 结构(TG-BTB)，通过利用热点程序反复执行的特性，在程序执行过程中提取记录每两条跳转分支指令之间的跳转间隔，只要分支指令未跳转，就可以避免访问 BTB，从而大大降低了 BTB 的查询能耗。通过在每个表项中存储分支目标在 BTB 中的索引号，在分支指令发生跳转时，不用通过标签比较就可以直接找到存储在 TG-BTB 的表项，进一步降低了 BTB 的查询功耗。

实验分析表明，与传统 BTB 相比，TG-BTB 增加了大约

(下转第 214 页)

[17] 邱菊.基于蚁群算法的软件测试用例生成方法研究[J].软件导刊,2011,10(3):73-74.

[18] LU H Q,CHEN L,SONG Y S,et al.. An improved crossover operator of genetic algorithm[J]. Journal of PLA University of Science and Technology,2007,8(3):250-253. (in Chinese)
卢厚清,陈亮,宋以胜,等.一种遗传算法交叉算子的改进算法[J].解放军理工大学学报,2007,8(3):250-253.

[19] KONG X L,WANG Y,JU A L,et al. An Improved Quantum Evolutionary Algorithm Based on Regulation Law of Hermone in Endocrine System[J]. Journal of Northwestern Polytechnical University,2011,29(6):978-983. (in Chinese)
孔晓琳,王毅,巨安丽,等.基于内分泌激素调节机制的量子进化算法[J].西北工业大学学报,2011,29(6):978-983.

[20] MIRZAAGHAEI M,PASTORE F,PEZZÈ M. Supporting Test Suite Evolution through Test Case Adaptation[C]// IEEE Fifth International Conference on Software Testing. 2012:231-240.

(上接第 201 页)

0.76%的 CPI,但 TG-BTB 比传统 BTB 减少了 80%的 BTB 查询功耗。TG-BTB 可以很容易地应用在其他流水线结构的处理器中,只需要在指令提取段和指令执行段更新临时寄存器、剩余轨迹长度计数器和跳转轨迹长度累加器的值。这种方法可以成为低功耗嵌入式处理器或高性能处理器降低功耗的技术。由于多核处理器架构设计师都把精力集中在降低单核的功耗上,因此这种降低单核功耗的技术显得尤为重要。

参 考 文 献

[1] PERLEBERG C,SMITH A. Branch target buffer design and optimizations[J]. IEEE Transactions on Computers,1993,42(4):396-412.

[2] MANNE S,KLAUSER A,GRUNWALD D. Pipeline gating:speculation control for energy reduction[C]// Proceedings of International Symposium on Computer Architecture. 1998:132-141.

[3] BONANNO J,COLLURA A,LIPETZ D,et al. Two level bulk preload branch prediction[C]// Proceedings of the IEEE International Symposium on High Performance Computer Architecture. 2013:71-82.

[4] KEETON K,PATTERSON D A,HE Y Q,et al. Performance-characterization of a Quad Pentium Pro SMP using OLTP workloads[C]// Proceedings of International Symposium on Computer Architecture. 1998:15-26.

[5] ANNAVARAM M,DIEP T,SHEN J. Branch behavior of a commercial OLTP workload on Intel IA32 processors[C]// Proceedings of International Conference on Computer Design. 2002:242-248.

[6] PYNE S,PAL A. Branch Target Buffer Energy Reduction Through Efficient Multiway Branch Translation Techniques[J]. Journal of Low Power Electronics,2012,8(5):604-623.

[7] HILGENDORF R B,HEIM G J,ROSENSTIEL W. Evaluation of branch-prediction methods on traces from commercial applications[J]. IBM Journal of Research and Development,1999,43:579-593.

[8] SUSSENGUTH E H. INSTRUCTION SEQUENCE CONTROL:US,US3559183[OL]. http://www.google.com/patents/US3559183.

[9] LEE J,SMITH A. Branch Prediction Strategies and Branch Target Buffer Design[J]. Computer,1984,17(1):6-22.

[10] CASZZA J. First the Tick,Now the Tock:Intel Microarchitecture[R]. Nehalem,2009.

[11] DRIESEN K,HÖLZLE U. The cascaded predictor:economical and adaptive branch target prediction[C]// Proceedings of International Symposium on Microarchitecture. 1998:249-258.

[12] FAGIN B,RUSSELL K. Partial resolution in branch target buffers[C]// Proceedings of International Symposium on Microarchitecture. 1995:193-198.

[13] KOBAYASHI R,YAMADA Y,ANDO H,et al. A Cost-Effective Branch Target Buffer with a Two-Level Table Organization [C]// Proceedings of International Symposium of Low-Power and High-Speed Chips. 1999:285-285.

[14] KAELI D R,EMMA P G. Branch history table prediction of moving target branches due to subroutine returns[C]// Proceedings of International Symposium on Computer Architecture. 1991:34-42.

[15] JOAO J A,MUTLU O,KIM H,et al. Improving the performance of object-oriented languages with dynamic predication of indirect jumps[C]// Proceedings of International Conference on Architectural Support for Programming Languages and Operating. 2008:80-85.

[16] SEZNEC A,FELIX V,KRISHNAN V,et al. Design tradeoffs for the Alpha EV8conditional branch predictor[C]// Proceedings of International Symposium on Computer Architecture. 2002:295-306.

[17] WANG G P,HU X D,YIN F,et al. Research and Design of Hash Indexing Mechanism for BTB[J]. Journal of Computer Research & Development,2014,51(9):2003-2011. (in Chinese)
王国澎,胡向东,尹飞,等. BTB 索引散列算法的研究与设计[J].计算机研究与发展,2014,51(9):2003-2011.

[18] ORAILOGLU A,PETROV P. Low-Power Data Memory Communication for Application-Specific Embedded Processors[C]// International Symposium on System Synthesis. 2002:219-224.

[19] PARIKH D,SKADRON K,ZHANG Y,et al. Power-Aware Branch Prediction:Characterization and Design[J]. IEEE Transactions on Computers,2004,53(2):168-186.

[20] SMITH J E,GOODMAN J R. A Study of Instruction Cache Organizations and Replacement Policies[J]. Acm Sigarch Computer Architecture News,1983,11(3):132-137.

[21] KAYNAK C,GROT B,FALSAFI B. Confluence:unified instruction supply for scale-out servers[C]// International Symposium on Microarchitecture. ACM,2015:166-177.

[22] DALLY W J,BALFOUR J,BLACK-SHAFFER D,et al. Efficient Embedded Computing[J]. Computer,2008,41(7):27-32.