第 33 卷 第 4 期 计 算 机 工 程 2007 年 2 月
Vol.33 No.4 Computer Engineering February 2007
·工程应用技术与实现· 文章编号：1000—3428(2007)04—0256—03 文献标识码：A 中图分类号：TP368.1

基于 EPIC 的同时多线程处理器取指策略

贾小敏，孙彩霞，张民选
（国防科学技术大学计算机学院，长沙 410073）

摘 要：EPIC 硬件简单，同时多线程易于开发线程级并行，在 EPIC 上实现同时多线程可以结合二者的优点。取指策略对同时多线程处理器的性能有重要影响。该文介绍了几种有代表性的超标量同时多线程处理器取指策略，分析了这些策略在 EPIC 同时多线程处理器上的适用性，提出了一种新的适用于 EPIC 的取指策略 SICOUNT。分析表明 SICOUNT 策略可以充分利用 EPIC 软硬件协同的优势，在选择取指线程时使用编译器所提供的停顿信息，能更精确地估计各个线程的流动速度，使取出指令的质量更高。

关键词：显式并行指令计算；同时多线程；取指策略；Itanium；SICOUNT

Instruction Fetch Policies for SMT Processors Based on EPIC

JIA Xiaomin, SUN Caixia, ZHANG Minxuan
（College of Computer, National University of Defense Technology, Changsha 410073）

【Abstract】 Explicitly parallel instruction computing (EPIC) can decrease the complexity of hardware, and simultaneous multithread(SMT) has the unique ability to exploit TLP(Thread Level Parallelism), so great benefit can be obtained by combining these two techniques. Fetch policies are of great importance to the overall performance of SMT processors. This paper describes several prevailing fetch policies used in superscalar SMT processors, analyses their applicability for EPIC SMT processors, and proposes a novel fetch policy called SICOUNT (Stop ICOUNT) suitable for EPIC SMT. Detailed analysis indicates that SICOUNT strategy can take full advantage of the HW/SW Co-design characteristic of EPIC. It achieves this by taking into account the stop hints generated by EPIC compiler. This makes it easier for SICOUNT to evaluate the flowing speed of each thread more precisely, and to fetch instructions with better quality.

【Key words】 Explicitly parallel instruction computing (EPIC); Simultaneous multithread(SMT); Fetch policy; Itanium; SICOUNT

同时多线程处理器[1,2]通过每个周期同时运行来自不同线程的指令来提高性能，可以在开发指令级并行(Instruction Level Parallelism, ILP)的同时，开发线程级并行(Thread Level Parallelism, TLP)。取指一直是同时多线程(Simultaneous Multithreading, SMT)研究的热点问题之一，因为它通常是同时多线程处理器性能的瓶颈所在[2]。

以往针对同时多线程的研究多以超标量为基础，但是超标量主要依赖硬件的动态调度来开发 ILP，可以开发的 ILP 有限，并且硬件复杂。显式并行指令计算(Explicitly Parallel Instruction Computing, EPIC)依据软、硬件的协同工作，把开发 ILP 的大部分工作交给编译器，硬件根据编译器的优化指导信息进一步开发 ILP，因而硬件简单，高效。在 EPIC 的基础上实现同时多线程，可以结合二者的优点，在硬件比较简单的情况下开发 TLP。

EPIC 与超标量在体系结构上相差甚远，以往针对超标量的同时多线程取指策略可能并不适用于 EPIC 同时多线程，因而，对 EPIC 同时多线程取指策略进行研究很有现实意义。

1 超标量同时多线程取指策略

在取指机制中，涉及到两个概念：取指带宽和取指吞吐量。取指带宽是指取指机构每个周期可以取出的最大指令数；而取指吞吐量则指每个周期实际取出的指令数。取指策略的目标就是在不增加取指带宽的情况下提高取指吞吐量。

基于超标量的同时多线程取指策略基本上都是通过对以下方面的改进来提高吞吐量的：(1)取指效率(Efficiency)，即取指带宽的利用率；(2)取出指令的有用性(Effectiveness)，即取出指令的质量[2]。本文重点介绍几种有代表性的取指策略。

1.1 提高取指效率

提高取指效率的方法主要是在多个线程间划分取指带宽。Tullsen 等在文献[2]中提出了几种可能的划分方法(假设取指带宽为 8，X 代表线程选择策略，可以为 RR(round-robin)以及后面提出的 ICOUNT 等)：

(1)X.1.8：每周期从 1 个线程取指，最多可取 8 条指令。

(2)X.2.4：每周期从 2 个线程取指，每个线程最多可取 4 条指令。

(3)X.4.2：每周期从 4 个线程取指，每个线程最多可取 2 条指令。

(4)X.2.8：每周期从 2 个线程取指，每个线程最多可取 8 条指令。

首先尽可能多地从第 1 个线程取指，如果不足 8 条，那么用第 2 个线程的指令补足。

模拟结果表明[2]X.2.8 的带宽利用率最好，这主要是因为 X.2.8 没有固定地将带宽划分给某个线程，而是按需分配，使得带宽可以尽可能地被利用。

1.2 提高取出指令的有用性

从两个方面衡量指令的有用性，即指令是否是有用的以及指令能否马上被执行。取指时，可以按照一定的准则来选出取出指令质量比较高的线程。

基金项目：国家“863”计划基金资助重大项目(2002AA110020)；国家自然科学基金资助项目(60273069)
作者简介：贾小敏(1982－)，女，硕士生，主研方向：新型微处理器体系结构；孙彩霞，博士生；张民选，教授、博导
收稿日期：2006-04-29 **E-mail：**echos5@126.com

Tullsen最初提出了几种线程选择策略[2]。

(1)BRCOUNT：把最高优先权分配给处于错误路径可能性最小的线程。为了实现这种策略，需要计算各个线程处于译码段、重命名段和指令队列中的分支指令个数，将最高的优先级赋予拥有未解决分支指令最少的线程。

(2)MISSCOUNT：一个长的存储器延时指令会导致与其相关的指令在指令队列中逗留很久，直到 Load 操作完成，从而一个线程阻塞后其后续指令把指令队列塞满。这种策略解决了这个问题，它把最高优先权赋予数据 Cache 失效最少的线程。

(3)ICOUNT：把最高优先权赋予处于译码、重命名和指令队列中指令最少的线程。这样做有 3 个好处：1)阻止任何线程塞满指令队列；2)它把最高优先权赋予能够通过指令队列最有效移动指令的线程；3)从可用的线程中提供比较混合的指令组合，使队列中的指令并行性最大化。

模拟表明[2]，ICOUNT的性能最好，现在的大多数超标量同时多线程取指策略都是在ICOUNT基础上进行改进的。

随着存储器和处理器速度差异的增大，长存储延迟如二级Cache失效对同时多线程的影响也越来越大。当一个线程的Load操作发生阻塞时，即使在乱序流出的处理器中，依赖于该操作的指令也会滞留在队列中，占用处理器的资源，迫使其他活跃线程无法全速流动。针对这个问题，Tullsen等以ICOUNT取指策略为基础，提出几种改进策略[3]来减少长延迟带来的性能损失。

(4)STALL：在 ICOUNT 的基础上进行改进，减少 Cache 失效率高的线程造成的影响。STALL 试图阻止发生 L2 Cache 失效的线程占用大量共享资源，它会检测一个线程是否拥有 L2 Cache 失效，如果有，就停止从该线程取指，但是不释放资源。

(5)FLUSH：是对 STALL 的扩展，发生 L2 Cache 失效的线程释放占用的所有资源，可以完全解决资源滥用问题。

(6)FLUSH++：FLUSH++是对 FLUSH 和 STALL 的折中，当工作负载对资源的需求压力不大时(L2 Cache 失效率高的线程较少)，STALL 的工作效果要优于 FLUSH，反之也成立。因此 FLUSH++根据 Cache 的行为在 FLUSH 和 STALL 之间进行切换。

2 EPIC 同时多线程取指策略

2.1 EPIC 机制对取指可能产生的限制

超标量通过硬件动态调度来开发 ILP，并且大都支持指令的乱序执行，无论是在取指、译码、还是分派阶段，对指令的处理都以单个的指令为单位。而 EPIC 则采用软硬件协同的思想，编译器负责指令的调度和优化，产生优化指导信息，指导硬件执行，硬件则采用简单的顺序发射、顺序执行模式，并且取指机构和指令队列的处理单位不再是指令，而是指令包(bundle)。

Itanium处理器取指带宽为 2 个指令包，分派带宽为 6 条指令；指令包队列以指令包为粒度向分派网络提供指令；分派网络则以指令为粒度向端口发射操作；一个指令包中的指令之间可以分离发射[5]。

指令包包含 3 个指令槽和一个模板字段，在代码生成阶段由编译器设置。指令模板体现了 EPIC 中编译器与硬件之间通信这一点，它包含了 3 条指令所需的资源信息和相关信息。硬件可根据资源信息把指令映射到相应的功能部件；相关信息则指出指令包中某条指令后需要一个停顿位 s，提示硬件在此之前有一条或多条指令与后面的一条或多条指令有相关，硬件可做出相应的处理。

EPIC 与超标量处理器之间的这些差异，对于在其上实现同时多线程，可能产生一些限制，主要有：

(1)取指带宽的“ 静态 ”划分

前面已经提过，在 Itanium 中取指粒度不是单个的指令，而是指令包，取指带宽为两个指令包。这就限制了取指带宽在多个线程之间的划分，虽然取指机构每周期可从指令 Cache 中取出 6 条指令，但是取指带宽却不能在 6 个线程之间划分，最多只能在两个线程之间划分，相当于已经将取指带宽“ 静态 ”地固定分配为两部分。而同时，指令包中的指令存在分离发射的可能，分离发射可导致该线程产生停顿，但分离发射的指令包中已经发射出去的指令还占据着指令包中的指令槽，本线程的指令因为相关和顺序发射的限制无法使用该指令包中的指令槽进行发射，而其他线程的指令虽与该线程的指令不相关，但由于指令包的限制，却不能使用分离发射槽的带宽，因此这种“ 静态 ”划分限制了 TLP 的开发。

(2)指令队列的设置

问题的关键是采用全局唯一的指令(包)队列还是为每一个活动线程设置一个指令(包)队列。在超标量中，采用全局唯一的指令队列是比较容易的，因为大多数超标量采用乱序发射的机制，加入同时多线程后并不需要增加复杂机制来选取指令进行分派。而在 EPIC 中，选择指令包进入分派网络的逻辑对原有硬件的改动会比较大，这是因为不同于超标量的乱序逻辑，Itanium 采用顺序执行方式，分派网络从指令包队列中取指令包时只是简单地从队列头取。但加入同时多线程后，如果一个线程的指令包阻塞在分派网络中，还可以从别的线程取指分派，这时对于单个指令包队列模式，就需要对这个指令包队列从头进行扫描，寻找未被阻塞的线程的第 1 个指令包，这与 EPIC 简化硬件的初衷相悖。另一个选择是为每个活动线程设置一个指令包队列。这样可以避免分派指令包时扫描整个指令包队列，但是需要线程选择逻辑来决定把哪个(哪些)指令包队列中的指令送入分派网络。

(3)顺序发射

由于处理器通过编译器来选择指令执行顺序，Itanium 中的流水线只能顺序发射指令，而不会对指令进行硬件动态重排序，因此在指令发射时任何指令的停顿都将会导致其所有后续指令停顿，一旦某一个线程有一条指令发生阻塞，则整个线程的后续指令包都将被阻塞在指令包队列中。顺序发射使得单个指令的阻塞或者停顿对处理器的吞吐量影响更大。因而在线程选择策略中应更多地考虑线程阻塞的影响，并且线程阻塞后的资源处理也显得更为重要。

2.2 现有取指策略的适用性

首先考虑 1.1 节介绍的几种取指策略。要提高取指效率，关键是在多个线程间有效地划分带宽。Itanium 微处理器每周期可从指令 Cache 中取出两个指令包，指令带宽的划分方法可以有下面几种(假设线程选择策略为 X)：

(1)X.1.2：每周期只从一个线程取指，每个线程取两个指令包。

(2)X.2.1：每周期从两个线程取指，每个线程最多可取指一个指令包。

(3)X.2.2：每周期可以从两个线程取指，每个线程最多取两个指令包。首先尽可能多地从第 1 个线程取指，如果第 1 个线程取不满两个指令包，那么从第 2 个线程补足。

在 X.1.2 和 X.2.1 中，取指带宽的划分是“静态”的，当有一个线程发生指令 Cache 失效无法取出指令时，别的线程也不能有效利用已经被静态划分给该线程的带宽。而在 X.2.2 中，取指带宽的划分不是固定的，总是尽可能地利用现有带宽取指，因而效率要高。

接下来考虑 1.2 节介绍的几种取指策略。

(1)MISSCOUNT 把最高优先权赋予数据 Cache 失效最少的线程。在 EPIC 中这种方法的优越性可能比在超标量同时多线程中更明显。这是因为 Itanium 是顺序发射、顺序执行的，一旦发生 Cache 失效，所有该线程的后续指令都会被阻塞，因而 Cache 失效对一个线程在处理器中的“流动速度”影响会更大。而 MISSCOUNT 的目标就是减少 Cache 失效带来的影响。

(2)BRCOUNT 将最高的优先级赋予拥有未解决分支指令最少的线程。这种方法对于 EPIC 同时多线程来说也是适用的，可以用来减少指令处于错误路径所带来的开销。这是因为 Itanium 支持硬件的前瞻执行，某一时刻处理器中会存在多个未解决的分支，可以以未解决分支的数目作为线程选择的标准。

(3)ICOUNT 把最高优先权赋予处于译码、重命名和指令队列中指令最少的线程。但是在 EPIC 中，这种方法需要进行一些改动，即指令包计数只考虑处于指令包队列中的指令包数，把最高优先权赋予指令包队列中指令最少的线程。虽然 ICOUNT 在 EPIC 中也适用，但可以预见，其作用将没有在超标量中那么显著，这是因为 EPIC 是顺序执行的，一旦一个线程的某个指令被阻住了，整个线程的后续指令包都不能流动，这样，即使该线程指令包队列中的指令包最少，该线程的流动速度也是很慢的。

(4)STALL 检测到一个线程有 L2 Cache 失效时，停止从该线程取指，但不释放资源。在 EPIC 中，STALL 的意义不大，这是因为 EPIC 是顺序执行的，一旦一个线程发生 L2 Cache 失效，整个线程的后续指令包都不能流动，这个线程的后续指令包有可能已经占据了宝贵的分派网络资源，使该资源无法为别的可分派线程的指令包利用。另外，EPIC 同时多线程中一般会采用分离的指令包队列，停止取指也不能使该线程的指令包队列为其他线程所用，因此 STALL 可以带来的性能提升很小。

(5)FLUSH 让发生 L2 Cache 失效的线程释放占用的所有资源。从对 STALL 的分析，可以看出，FLUSH 对 EPIC 来说更有意义，因为它可以完全解决资源滥用问题，但是由于需要对该线程重新取指，存在功耗增加的问题。

(6)FLUSH++根据 Cache 的行为在 FLUSH 和 STALL 之间进行切换。但是依据对 STALL 和 FLUSH 的分析，选择 FLUSH 意义更大。而当线程比较少，别的线程并不需要 FLUSH 所释放的资源时，不做处理即可。

2.3 一种新的取指策略

(1)基础结构

前面已经提到过，在 EPIC 中，采用每个线程分离的指令包队列效率要高一些，可以省去复杂的扫描队列查找可执行线程指令包的过程。因此采用分离的指令包队列作为这种新的取指策略的基础。但是，这样又会产生新的问题，那就是如何从多个线程中选择指令包进入分派网络。分派网络的线程选择策略显然也有多种选择，在这里选择轮转(Round Robin)模式作为基本模式，选择轮转模式的初衷是试图使单个线程的指令包之间的执行距离尽可能远，从而减少指令包之间的相关引起的停顿。

所以，采用分离的指令包队列、轮转的分派线程选择策略(如图 1)作为基础结构，同时采用 2.2 节提到的取指带宽划分机制 X.2.2。

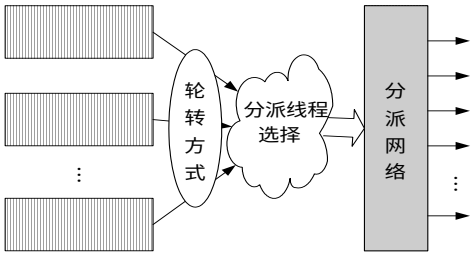

图 1 轮转式分派线程选择策略

(2)SICOUNT 策略

基于这个结构，可以提出一种新的基于 Itanium 微体系结构的同时多线程选择策略：SICOUNT(Stop ICOUNT)。

从名字上可以看出这种方法是在 ICOUNT 的基础上进行改进的。2.2 节提到，由于 EPIC 的顺序执行，ICOUNT 在 EPIC 中的作用将没有在超标量中那么显著，一个线程的某个指令阻塞将导致其所有后续指令包都不能流动，这样，即使该线程指令包队列中的指令包最少，该线程的流动速度也是很慢的。因而 ICOUNT 并不能对 EPIC 中的线程流动速度做出精确的估计。

SICOUNT 的目标是充分利用 EPIC 中编译器和硬件的通信能力，通过在线程选择策略中对编译器提供的相关信息加以考虑，使取指机构能更精确地估计各个线程的流动速度。

具体方法是，决定线程优先级的计数部分由两部分组成：1)各个线程指令包队列中的指令包的个数；2)各个线程的指令包队列中所有指令包所包含的停顿位总数再乘上一个因子。即

$$C_k = B_{k,BQ} + \lambda S_{k,BQ}$$

其中，C_k 代表线程 k 的计数值，而 $B_{k,BQ}$、$S_{k,BQ}$ 分别代表线程 k 在指令包队列(Bundle Queue)中的指令包总数以及这些指令包中包含的停顿位(Stop)总数。λ 称为停顿因子，取值在 1 左右，表示指令包中包含的停顿位(Stop)总数在整个优先权计数中所占的权重。

SICOUNT 在选择线程进行取指时，把最高优先权赋给计数值 C_k 最小的线程。下面分析一下 SICOUNT 的工作机理。

我们注意到 EPIC 的编译器提供一定的优化指导信息给执行硬件，从前面对 Itanium 指令模板的介绍中知道，编译器会在指令包的指令模板字段提供指令间的相关信息，指示某一个指令后需要一个停顿位 s。SICOUNT 线程选择策略的提出正是基于对 EPIC 这一特点的观察。它基于这样一种假设，就是大多数拥有停顿位的线程在执行时都至少会停顿一个周期，从而使单个指令包的执行时间相对增加一个周期，这相当于是两个没有任何停顿的指令包重叠执行的时间。当然，这种假设是对 EPIC 复杂的执行机制的一种简化。但可以看出这种假设在某种意义上是合理的：虽然采用轮转模式的分派线程选择策略，已经很大程度上减弱了指令包之间的相关带来的影响，但是，轮转模式并不能减弱一个指令包内部的停顿带来的周期损失。因此指令包中的停顿位总会引起该指令包分离发射，导致该指令包的分派至少需要 1+λ 个周

(下转第 262 页)