

使用取指策略控制同时多线程处理器中 个体线程的性能

孙彩霞 张民选

(国防科学技术大学计算机学院 长沙 410073)

摘 要 当前,对同时多线程(Simultaneous Multithreading, SMT)处理器取指策略的研究大都集中在总体性能的优化上. 文中提出一种新颖的 SMT 处理器取指策略(Controlling Performance of Individual Thread, CPIT),用于控制个体线程的执行. 结果表明,对于模拟的所有负载,CPIT 在 94% 以上的情况下都能保证受控线程获得期望性能. 而对于失败的情况,受控线程的平均性能偏差不超过 1.25%. 此外,CPIT 策略对处理器总体性能的影响并不大. 与 ICOUNT 这种以优化性能为目标的取指策略相比,总体性能的平均降低不超过 3%,而除受控线程外的其他线程的性能平均只降低了 1.75%.

关键词 同时多线程;取指策略;性能;资源分配;期望性能

中图法分类号 TP303

Using Instruction Fetch Policy to Control Performance of a Thread in SMT Processors

SUN Cai-Xia ZHANG Min-Xuan

(School of Computer Science, National University of Defense Technology, Changsha 410073)

Abstract Currently, fetch policies in Simultaneous Multithreading (SMT) processors almost focus on overall performance optimization, and provide no control over how individual threads are executed. A novel fetch policy called CPIT (Controlling Performance of Individual Thread) is proposed to control the execution of a particular thread in SMT processors. Results show that for more than 94% of all cases measured, CPIT can control the execution and consequently achieve the desired performance for a given thread. For the failing cases, the average variance is within 1.25%. Furthermore, CPIT does not sacrifice overall performance of SMT processors severely. Compared to fetch policies orienting towards performance maximization such as ICOUNT, the average degradation of overall performance is not more than 3% and the degradation of threads other than the given thread in performance is only 1.75%.

Keywords Simultaneous multithreading; instruction fetch policy; performance; resource allocation; desired performance

1 引 言

在同时多线程(Simultaneous Multithreading,

SMT)处理器^[1-3]中,同时运行的线程共享处理器资源,主要包括发射队列、物理寄存器、执行单元和再定序缓冲(Reorder Buffer, ROB)等. 共享资源在线程之间的分配方式对 SMT 处理器的总体性能和每

个线程的性能都具有决定性的影响。目前, SMT 处理器中的资源分配主要由取指策略决定。然而, 以前对取指策略的研究几乎都是针对如何提高总体性能的, 如 ICOUNT^[2], STALL^[4], FLUSH^[4], DG^[5] 以及 PDG^[5] 等。这些策略都是以优化总体性能为目标的, 导致个体线程的性能随取指策略的不同变化很大。

另一方面, SMT 处理器中同时运行的线程也在竞争资源, 并且不同的线程竞争资源的能力也大不一样。因此, 对于同一个线程, 当和不同的线程组成负载运行在 SMT 处理器上时, 竞争到的资源数目会发生变化, 从而导致该线程的性能也会发生变化。

图 1 给出了 crafty 单独运行以及和不同线程同时运行时的性能, 这里用 IPC 作为性能度量标准。当和其他线程同时运行时, 取指策略分别采用 RR (轮转法, ROUND-ROBIN) 和 ICOUNT。可以看到, crafty 的性能随取指策略和负载的不同而变化很大。

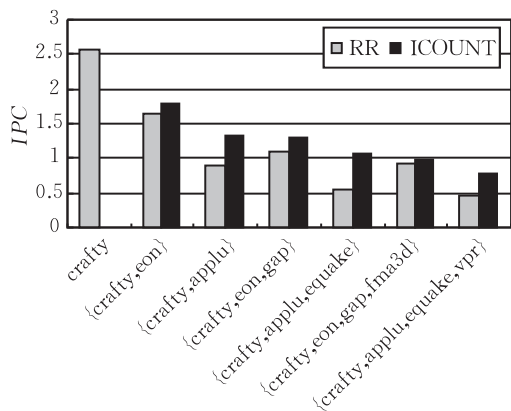


图 1 不同取指策略和不同负载组合下 crafty 的性能

在单线程处理器系统中, 操作系统线程调度器通过分配时间片可以控制个体线程的执行, 能够保证某个线程在规定的时间内完成。然而, 对于 SMT 处理器系统, 在任意一个时间片内, 处理器中个体线程的性能都是不可预测的, 这种性能的不可预测性意味着个体线程需要的执行时间是不可预测的, 因此, 单线程处理器系统所采用的时间片分配法无法保证 SMT 处理器中个体线程在规定时间内完成。但是, 有些时候我们需要保证个体线程的执行时间满足一定的要求, 尤其是对于实时系统和嵌入式系统, 对实时性要求非常严格, 某些任务必须在规定时间内完成, 否则系统就将不能正常工作。

把执行时间有限制的线程用 CT (Critical Thread) 表示, 而和 CT 同时运行的其他线程用 NCT (Non-Critical Thread) 表示。在本文的研究中, CT 只有一

个。为了保证 CT 在规定时间内完成, 最简单的办法是让 CT 在 SMT 处理器上单独运行。但是这将意味着同时多线程体系结构失去了意义。我们希望在保证 CT 的执行时间满足要求的同时, 尽可能充分利用 SMT 中的资源提高 NCT 的性能和总体性能。为此, 除了操作系统线程调度器对线程进行调度外, 硬件必须提供相应的支持。本文提出使用取指策略控制 CT 的执行, 使 CT 在任意一个时间片内都能获得期望的性能, 从而实现 CT 的执行时间可控, 而不受负载组合中 NCT 的影响, 这种新的取指策略叫 CPIT (Controlling Performance of Individual Thread)。CPIT 的目标是在保证 CT 获得期望性能的前提下, 尽可能提高 NCT 的性能和总体性能。

本文第 2 节介绍相关研究; 第 3 节详细描述 CPIT 策略以及如何使用 CPIT 控制 CT 的性能; 第 4 节给出模拟环境、测试程序以及相关参数的设置; 第 5 节给出实验结果并对结果进行分析; 最后, 在第 6 节总结全文。

2 相关工作

当前的取指策略几乎都是针对如何提高总体性能的, ICOUNT 就是其中最具有代表性的一种, 并被广泛应用于同时多线程处理器中。ICOUNT 策略的基本思想是优先从处于译码段、重命名段和发射队列中指令数目最少的线程取指。基于 ICOUNT, 还提出了许多取指策略, 如 STALL, FLUSH, DG 和 PDG 等。为了提高总体性能, 这些策略通常会让固有 IPC 较高的线程优先执行, 而并不关心个体线程的执行。

不过人们已经开始关注 SMT 处理器上个体线程性能的不可预测这个问题。在文献[6]中, Anantaraman 提出: 如果 SMT 处理器上的一个任务在多线程模式下无法在规定时间内完成, 那么就切换到单线程模式, 单独运行这个任务。我们前面已经说过, 这种方法会浪费 SMT 中的处理器资源。

在文献[7]中, Snavely 等提出了一种扩展的 ICOUNT 策略: 每个线程被赋予一个可以反映其优先权的数值, 根据这个数值决定线程的取指优先级。这种方法可以保证各个线程获取与优先权一致的性能, 可是具体性能仍然是无法预测的。

在文献[8-10]中, Cazorla 等提出一种硬件机制: 由资源分配器显式地决定个体线程可以占用的资源数目。通过不断调整分配给 CT 的资源数目使得 CT 可以获得期望性能。这种机制可以达到控制个体线程性能的目的, 但是硬件实现非常复杂。如用

于调整资源分配的资源分配器,多个用于监视各个线程资源使用情况的资源使用计数器,等等.该机制的具体实现和硬件开销请参见文献[10].

3 CPIT 取指策略

3.1 基本思想

CPIT 取指策略的基本思想是直接监测 CT 的性能,并和期望性能进行比较,根据比较结果调整 CT 的取指优先级,从而控制 CT 的性能.

CPIT 实际上是一种混合的取指策略,它把 ICOUNT2.8 作为缺省的策略. ICOUNT2.8 的含义是:各个线程的取指优先级由 ICOUNT 策略决定,每个周期最多可以从 2 个线程取指,每个周期最多可以取出 8 条指令.此外,还定义了两种新的策略:PCT(Prioritizing Critical Thread)和 PNCT(Prioritizing Non-Critical Thread).

PCT:CT 的取指优先级最高,其他线程(即 NCT)的取指优先级由 ICOUNT 决定.

PNCT:CT 的取指优先级最低,其他线程(即 NCT)的取指优先级由 ICOUNT 决定.

令 IPC_{dsr} 表示 CT 期望获得的性能, IPC_{real} 表示 CT 实际获得的性能, D_{IPC} 由式(1)定义:

$$D_{IPC} = IPC_{real} - IPC_{dsr} \quad (1)$$

CPIT 策略根据 D_{IPC} 的值在 PCT2.8, ICOUNT2.8 和 PNCT2.8 之间进行切换.如果 D_{IPC} 小于某个阈值 $Th_{PCT2.8}$,表示 CT 的实际性能离期望的值已经有一定的差距,所以切换到 PCT2.8 以加速 CT 的执行;如果 D_{IPC} 大于某个阈值 $Th_{PNCT2.8}$,表示 CT 的实际性能已经超出期望的值,切换到 PNCT2.8,通过让 NCT 优先取指来提高 NCT 的性能;如果 D_{IPC} 的值介于 $Th_{PCT2.8}$ 和 $Th_{PNCT2.8}$ 之间,表示 CT 的实际性能和期望的值相差无几,所以使用 ICOUNT2.8 作为取指策略,以更加有效地利用处理器资源,优化处理器的总体性能.

当使用 PCT2.8 加速 CT 的执行时可能会失败,即 CT 可能仍然无法获得期望的性能,尤其是当期望性能较高时,这种可能性会更大.这是因为,虽然 CT 具有最高的取指优先级,但是 PCT2.8 每个周期可以从 2 个线程取指,当 CT 无法填满整个取指带宽时就会从 NCT 取指.从 NCT 取指就意味着 NCT 要占用一定的共享资源,从而可能导致 CT 无法获得足够的资源以实现期望性能.为了解决这个问题,定义阈值 $Th_{PCT1.8}$,当 D_{IPC} 小于 $Th_{PCT1.8}$ 时,使

用 PCT1.8 作为取指策略,以阻止 NCT 占用更多的共享资源.

当只有两个线程同时运行时,使用 PNCT2.8 试图提高 NCT 的性能也可能会失败.因为只有两个线程同时运行,除了 CT 外只有一个 NCT,因此当 NCT 无法填满整个取指带宽时就会从 CT 取指,导致 CT 的实际性能远远超过期望的值,而 NCT 的性能却很低,这与 CPIT 的目标是相悖的.因此,定义阈值 $Th_{PNCT1.8}$,只有两个线程同时运行并且 D_{IPC} 大于 $Th_{PNCT1.8}$ 时,使用 PNCT1.8 取指,在保证 CT 获得期望性能的前提下,尽可能把共享资源留给 NCT.

获取 D_{IPC}

为了在不同的策略之间切换以调整 CT 的取指优先级,必须知道 D_{IPC} ,即必须知道 CT 的 IPC_{real} 和 IPC_{dsr} .我们把 IPC_{dsr} 表示成 CT 全速运行时性能的百分比,用 X 表示^[8],CT 全速运行时的性能即 CT 单独在 SMT 处理器上运行时的 IPC,用 IPC_{alone} 表示.因此, D_{IPC} 可以进一步表示成式(2)的形式:

$$D_{IPC} = IPC_{real} - X \times IPC_{alone} \quad (2)$$

假设操作系统知道 CT 需要获取的性能,即操作系统提供 X 的值; IPC_{real} 可以在 CT 和 NCT 同时运行过程中动态获取;只有 IPC_{alone} 不容易得到.为了动态获取 IPC_{alone} ,需要在 SMT 处理器上单独运行 CT.因此采用两个阶段:采样阶段(sample phase)和调整阶段(tune phase)^[8].

在采样阶段,处理器运行在单线程模式下,CT 在处理器上单独运行一段时间.采样阶段由两个子阶段组成.第一个子阶段是 warm-up 阶段,CT 虽然单独运行在处理器上,但是并不统计 IPC,这个子阶段是为了减轻 NCT 对共享资源的“污染”,尤其是对 cache 的“污染”,使得采样阶段获得的 IPC_{alone} 比较准确.第二个子阶段是实际采样阶段,对 CT 的执行情况进行统计,这个子阶段结束后,通过计算可以获得 IPC_{alone} .

在调整阶段,处理器运行在多线程模式下,CT 和其他 NCT 同时运行.每个周期,通过计算获得 IPC_{real} 和 D_{IPC} ,并根据 D_{IPC} 的值调整 CT 的取指优先级.

由于程序在不同的执行阶段 IPC 会有很大的差别^[11],为了在采样阶段获得的 IPC_{alone} 能够比较准确地代表 CT 在调整阶段的全速运行速度,我们交替执行采样阶段和调整阶段,即所有的线程同时运行一段时间(一个调整阶段)后重新采样,用新的

采样结果指导下一个调整阶段的执行. 为此, 需要为每个阶段定义一个参数, 表示该阶段持续的时间. 分别用 $L_{\text{warm-up}}$, $L_{\text{actual-sample}}$ 和 L_{tune} 表示 warm-up 阶段、实际采样阶段和调整阶段的长度, 单位是处理器时钟周期.

3.2 实 现

(1) D_{IPC} 的简化计算

在调整阶段, 每个周期都要更新 D_{IPC} . 由式(1)可知, 首先需要有一个除法操作计算 IPC_{real} , 因为每个周期 IPC_{real} 都会发生变化; 而 IPC_{dsr} 在采样阶段结束后就计算完毕, 在调整阶段可以直接使用; 得到 IPC_{real} 后, 还需要一个减法操作. 因此, 共需要串行执行一个除法和减法, 而且都是浮点操作. 在实现 CPIT 时, 对式(1)进行一定的变换, 以减少 D_{IPC} 的计算复杂度和计算时间.

令 INST 表示调整阶段经过 CYC 个时钟周期后 CT 已经提交的指令总数, 那么 CT 在过去的 CYC 个周期的 IPC_{real} 可以由式(3)给出:

$$\text{IPC}_{\text{real}} = \frac{\text{INST}}{\text{CYC}} \tag{3}$$

因此, D_{IPC} 可以表示成式(4)的形式:

$$D_{\text{IPC}} = \frac{\text{INST}}{\text{CYC}} - \text{IPC}_{\text{dsr}} \tag{4}$$

把式(4)中等式的左右两边同时乘以 CYC , 就得到式(5):

$$\begin{aligned} D'_{\text{IPC}} &= D_{\text{IPC}} \times \text{CYC} = \text{INST} - \text{IPC}_{\text{dsr}} \times \text{CYC} \\ &= \text{INST} - \text{INST}_{\text{dsr}}(\text{CYC}) \end{aligned} \tag{5}$$

式(5)中的 $\text{IPC}_{\text{dsr}} \times \text{CYC}$ 实际上就是为了达到

期望性能, CT 在 CYC 个时钟周期里应该执行的指令数目, 用 $\text{INST}_{\text{dsr}}(\text{CYC})$ 表示. $\text{INST}_{\text{dsr}}(\text{CYC})$ 具有这样的性质: 每个周期增加 IPC_{dsr} . 因此 $\text{INST}_{\text{dsr}}(\text{CYC})$ 可以通过累加得到, 并不需要使用乘法操作, 如式(6)所示:

$$D'_{\text{IPC}} = \text{INST} - (\text{INST}_{\text{dsr}}(\text{CYC}-1) + \text{IPC}_{\text{dsr}}) \tag{6}$$

所以计算 D'_{IPC} 需要串行执行一个加法和一个减法, 但还是浮点操作. 为此, 我们进一步简化, 把 IPC_{dsr} 用整数表示: 假设在实际采样阶段, CT 共执行了 Y 条指令, 先把 Y 和期望百分比 X 相乘, 得到的结果 Z 用整数表示. 在选择 $L_{\text{actual-sample}}$ 时总是令其可以表示成 2^k 的形式, 这样 Z 右移 k 位就可以得到 IPC_{dsr} . 但是 IPC_{dsr} 的精度会大大受损. 为此, 在计算 IPC_{dsr} 时, Z 只右移 $(k - N_{\text{shift}})$ 位, 也就是比正常情况少右移 N_{shift} 位, 把得到的结果用 IPC'_{dsr} 表示, 这样相当于式(6)的左右两边都左移 N_{shift} 位, 由此得到式(7), 其中 N_{shift} 是 CPIT 策略中新引入的一个参数, 表示计算 IPC_{dsr} 时少右移的位数:

$$\begin{aligned} D''_{\text{IPC}} &= D'_{\text{IPC}} \ll N_{\text{shift}} \\ &= (\text{INST} \ll N_{\text{shift}}) - (\text{INST}'_{\text{dsr}}(\text{CYC}-1) + \text{IPC}'_{\text{dsr}}) \end{aligned} \tag{7}$$

可见, 计算 D''_{IPC} 需要一个左移位、一个整数加法和整数减法, 并且加法操作和移位操作可以并行执行. 在 CPIT 策略中, 用 D''_{IPC} 代替 D_{IPC} 指导资源分配, 这样可以大大缩短指导资源分配的信息的滞后时间.

(2) 硬件实现

图 2 给出了 CPIT 策略的实现框图. 在采样阶

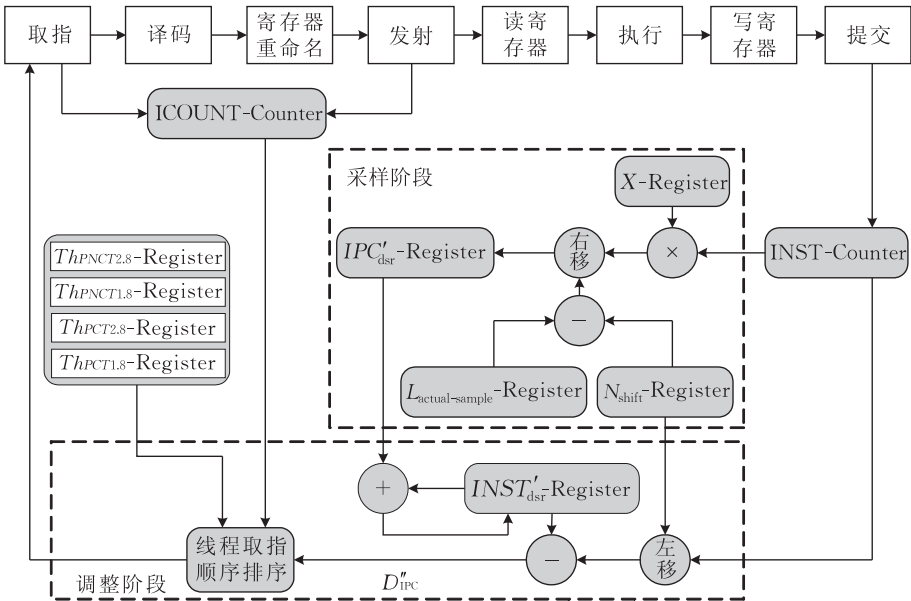


图 2 CPIT 策略的实现

段和调整阶段都需要记录 CT 已经提交的指令数目,因此需要一个计数器 INST-Counter,每提交一条 CT 的指令,INST-Counter 就会加 1. X-Register 寄存器保存的是 CT 期望获得的性能百分比; $L_{\text{actual-sample}}$ -Register 寄存器保存的是 $\log_2 L_{\text{actual-sample}}$, 由于选择实际采样阶段的长度时令其为 2^k 的形式, 所以该寄存器中的值实际上就是 k . 采样阶段结束后, INST-Counter 首先和 X-Register 中的值相乘并取整,然后再右移即可得到 IPC'_{dsr} , 存放在寄存器 IPC'_{dsr} -Register 中. 如果右移操作后得到的结果为 0, 那么把 1 写入 IPC'_{dsr} -Register 中. 进入调整阶段, INST-Counter 重新从 0 开始计数,记录 CT 已经提交的指令数目. $INST'_{\text{dsr}}$ -Register 保存 CT 应该执行的指令数目,每个时钟周期增加 IPC'_{dsr} 条指令,一个新的调整阶段开始时 $INST'_{\text{dsr}}$ -Register 重新从 0 开始累加. INST-Counter 左移后的值和 $INST'_{\text{dsr}}$ -Register 的差值作为策略切换条件,通过与切换阈值进行比较,决定如何对各个线程的取指顺序进行排序.

由于缺省策略是 ICOUNT, 所以每个线程需要一个计数器 (ICOUNT-Counter). 图中除了 ICOUNT-Counter 是每个线程都拥有外,其他的寄存器和计数器 NCT 并不需要,只有 CT 才拥有.

4 实验设置

我们采用 SMTSIM^[12] 进行实验模拟. SMTSIM 模拟了所有类型的延时,包括 Cache 访问延时、分支误预测延时、TLB 失效处理延时等. 同时, SMTSIM 可以收集和统计实验中有关的信息,如 Cache 失效

率,分支预测精度以及 IPC 等,在本文的实验中,我们只关心线程的 IPC 值. 表 1 给出了模拟器的基本配置.

表 1 模拟器的基本配置

参数	值
取指带宽	8 条指令/周期
指令队列	64 项整型队列,64 项浮点队列
功能单元	6 个整型(4 个 load/store),3 个浮点
重命名寄存器	100 个整数,100 个浮点
分支预测器	2K 项的 gshare
分支目标缓冲	256 项,4 路组相联
L1I Cache	64KB,2 路组相联,每行 64 字节,1 个周期访问延时
L1D Cache	64KB,2 路组相联,每行 64 字节,1 个周期访问延时
L2 Cache	512KB,2 路组相联,每行 64 字节,10 个周期访问延时
L2 Cache	4MB,2 路组相联,每行 64 字节,20 个周期访问延时
主存	100 个周期访问延时

表 2 给出了实验中所使用的所有测试程序,这些程序都来自 SPEC2000 测试集^①,并且都使用 reference 输入数据集. 根据 Cache 行为可以把我们使用的测试程序分成两组:一组是存储器访问密集的程序,这些程序的特点是 Cache 失效很多,在所模拟的指令片段中,平均每条指令遇到的 L2 Cache 失效数目大于 0.01;另一组是指令级并行性 (Instruction Level Parallelism, ILP) 较高的程序,这组程序的特点是 Cache 失效率较低. 在实验中,完整的模拟测试程序要花费大量的时间,因此我们采用文献[13]中所提到的方法,只模拟每个程序中最有代表性的指令片段. 表 2 的第 4 列给出了每个线程快速前进的指令数目.

表 2 测试程序

类型	测试程序	输入集	快速指令数目/亿	L2 Cache 失效个数/指令
存储器访问密集型 (MEM)	parser	ref	16	0.011
	twolf	ref	31	0.015
	lucas	ref	35	0.021
	art	c756hel.in(ref)	67	0.072
	swim	ref	5	0.041
	applu	ref	7	0.019
高 ILP 型 (ILP)	gzip	input.graphic(ref)	3	1.3E-4
	eon	cook (ref)	3	1.0E-4
	gap	ref	3	5.0E-4
	crafty	ref	0	9.0E-4
	fma3d	ref	1	2.4E-6
	mesa	ref	3	5.3E-4

CT 可以是 ILP 型程序也可以是 MEM 型程序, NCT 也是如此, 因此一个多线程负载由 3 个因素决定: CT 的类型、NCT 的类型和线程数目. 比如, IM3 型负载表示 CT 是 ILP 型程序, NCT 都是

MEM 型程序,共包含 3 个测试程序. 表 3 给出了实验中使用的多线程负载. 每个负载中的第一个线程

① <http://www.specbench.org>

是 CT,当 CT 运行完 3 亿条指令后结束模拟.为了 种类型的负载随机抽取了多组,最终结果取多组测试程序运行结果的平均值.

表 3 多线程负载

类型	包含的测试程序
II2	{crafty, eon}, {fma3d, gap}, {mesa, gzip}
IM2	{crafty, applu}, {fma3d, earthquake}, {mesa, twolf}
MI2	{vpr, eon}, {lucas, gap}, {swim, gzip}
MM2	{vpr, applu}, {lucas, earthquake}, {swim, twolf}
II3	{crafty, eon, gap}, {fma3d, gap, mesa}, {mesa, gzip, crafty}
IM3	{crafty, applu, lucas}, {fma3d, earthquake, swim}, {mesa, twolf, vpr}
MI3	{vpr, eon, gap}, {lucas, gap, mesa}, {swim, gzip, crafty}
MM3	{vpr, applu, lucas}, {lucas, earthquake, swim}, {swim, twolf, vpr}
II4	{crafty, eon, gap, fma3d}, {fma3d, gap, mesa, gzip}, {mesa, gzip, crafty, eon}
IM4	{crafty, applu, lucas, twolf}, {fma3d, earthquake, swim, vpr}, {mesa, twolf, vpr, earthquake}
MI4	{vpr, eon, gap, fma3d}, {lucas, gap, mesa, gzip}, {swim, gzip, crafty, eon}
MM4	{vpr, applu, lucas, twolf}, {lucas, earthquake, swim, vpr}, {swim, twolf, vpr, earthquake}

表 4 给出了在第 5 节的实验中 CPIT 取指策略中的参数的取值. 这些值都是通过理论分析和大量的实验得到的.

表 4 CPIT 取指策略中参数的值

参数	值
$Th_{PCT1.8}$	0
$Th_{PCT2.8}$	0
$Th_{PNCT2.8}$	2^{28}
$Th_{PNCT1.8}$	2^{32}
$L_{warm-up}$	2^{16}
$L_{actual-sample}$	2^{14}
L_{tune}	2^{21}
N_{shift}	17

5 实验结果

首先给出 CT 的性能,衡量 CPIT 取指策略在控制 CT 执行方面的优劣;然后给出 NCT 的性能和总体性能,并和 ICOUNT 取指策略的相应结果进行比较,衡量 CPIT 在控制个体线程性能时对总体性能和其他线程性能的影响.

5.1 CT 的性能

图 3 给出了对不同类型的负载使用 CPIT 策略得到的 CT 的性能. 横坐标除了给出负载中线程的数目外,还给出了 CT 的期望性能. 期望性能被表示为 CT 单独运行在 SMT 处理器上时性能的百分比,从 10%~90%不等. 纵坐标给出的是 CT 的实际性能与其单独运行在 SMT 处理器上时性能的比值.

可以看出,当百分比比较低时,对于所有类型的负载,CPIT 策略都能保证 CT 刚好获得期望的性能或者稍稍超出. 而当百分比达到 70%以及更高时,实际性能通常都要低于期望性能. 这是因为,采样阶段得到的 IPC_{alone} 通常并不能准确地代表调整阶段 CT 的全速运行速度,因此使用 IPC_{alone} 计算得到的期望性能(即 IPC_{dsr})与真正的期望性能(用 IPC'_{dsr} 表示)存在差别. 如果 $IPC_{dsr} < IPC'_{dsr}$,那么无论百分比是高还是低,在调整阶段 CT 的实际性能都可以达到 IPC_{dsr} ,也就是实际性能与 IPC'_{dsr} 相差的幅度都可以达到 $(IPC'_{dsr} - IPC_{dsr})$;而当 $IPC_{dsr} > IPC'_{dsr}$ 时,如果百分比很高,CT 的实际性能却不一定总能达到

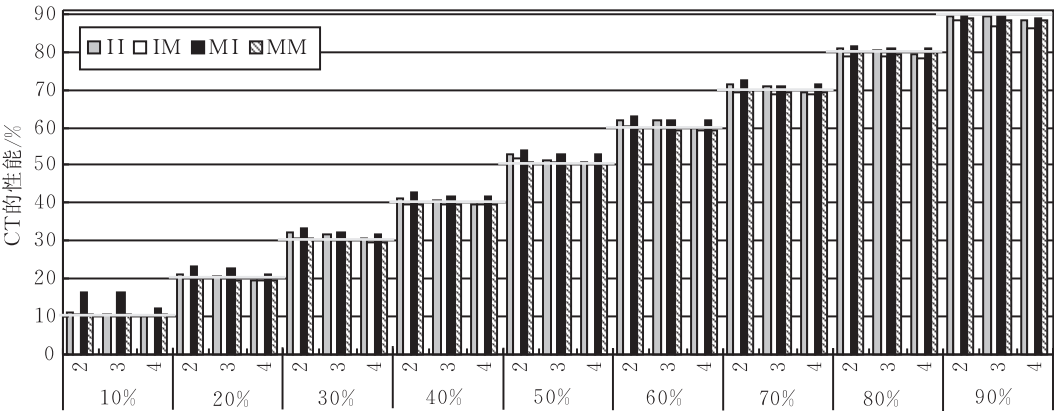


图 3 CPIT 取指策略下 CT 的性能

IPC_{dsr} . 比如,如果采样得到的 CT 的 IPC_{alone} 为 4, 百分比是 90%, 则 IPC_{dsr} 将是 3.6. 而 CT 在调整阶段的全速运行速度只有 3.5, 那么在调整阶段即使在 SMT 处理器上单独运行 CT, CT 的实际性能也无法达到 3.6. 当然, CT 的实际性能还会大于 IPC'_{dsr} , 但是高出的幅度却无法达到 $(IPC_{dsr} - IPC'_{dsr})$. 由此可见, 如果目标百分比比较低, 那么 CT 的实际性能总是能够达到 IPC_{dsr} , 通过互补作用, CT 最终能够获得真正的期望性能. 而当目标百分比比较高时, 这种互补作用减弱, 所以出现了图 3 所示的结果.

从图 3 还可以看出, 对于同样的 CT 和线程数目, 与 NCT 是 MEM 型程序的负载相比, NCT 是 ILP 型程序的负载获得的实际性能要高一些. 这是因为, 如果 NCT 是 MEM 型的程序, 它们对 L2 cache 造成的污染要比 ILP 型程序更加严重. 虽然我们试图通过 warm-up 阶段减轻这种污染, 但是却无法全部清除^[8]. 因此, 与 NCT 是 ILP 型程序的负载相比, 当 CT 和 MEM 型的 NCT 同时运行时,

在采样阶段得到的 IPC_{alone} 要相对小一些, 从而导致 CT 的实际性能也偏低.

总的来说, 对于模拟的所有负载, CPIT 在 94% 以上的情况下都能控制 CT 的执行, 使该线程达到期望性能. 而对于失败的情况, CT 的平均性能偏差也不超过 1.25%. 从上面的分析知道导致失败的原因主要有两个: (1) 在采样阶段得到的 IPC_{alone} 不能准确地代表 CT 在调整阶段的全速运行速度; (2) 由于 NCT 对共享资源尤其是 L2 cache 的污染, 导致采样阶段得到的 IPC_{alone} 偏小. 在本文, 我们只是通过 warm-up 阶段来部分减轻 NCT 对共享资源的污染. 在今后的工作中, 我们会继续对 CPIT 策略进行优化, 使采样得到的 IPC_{alone} 更加准确.

5.2 NCT 的性能

图 4 给出了与 ICOUNT 相比, CPIT 获得的总体性能. 纵坐标表示 CPIT 策略下的总体性能与 ICOUNT 策略下总体性能的比值. 同样的, 图 5 给出了与 ICOUNT 相比, CPIT 策略下 NCT 的性能.

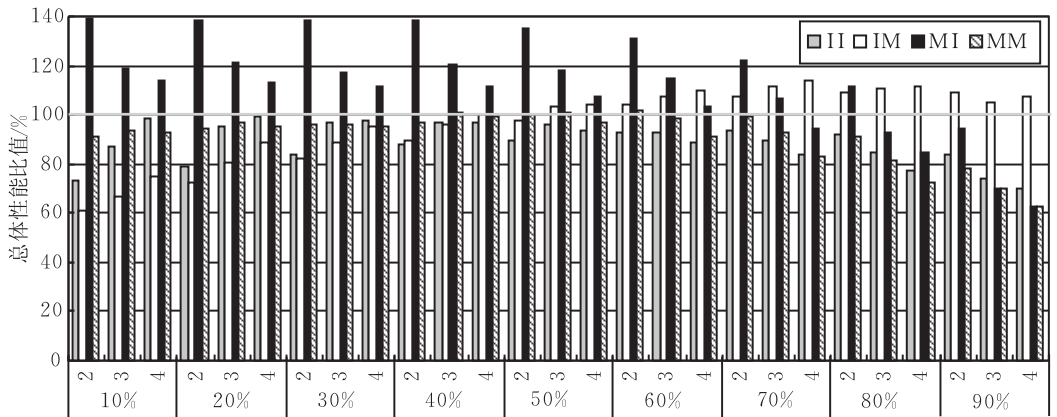


图 4 与 ICOUNT 相比, CPIT 获得的总体性能

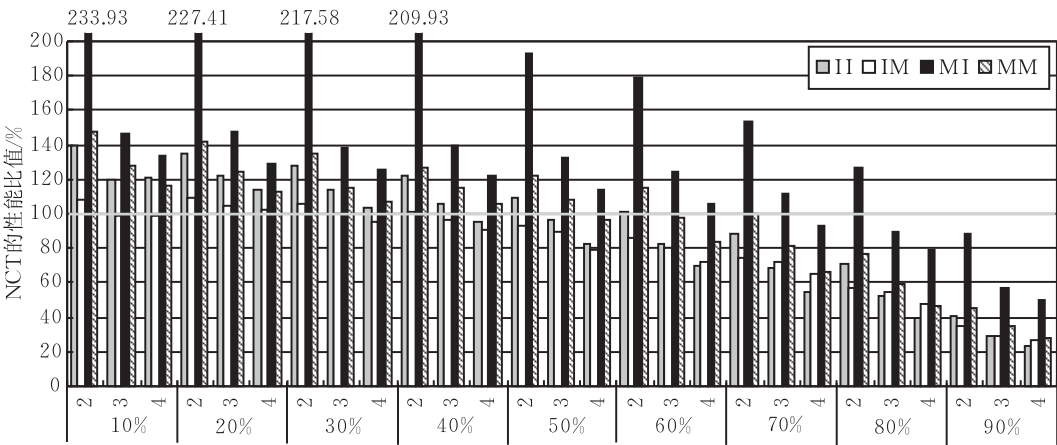


图 5 与 ICOUNT 相比, CPIT 取指策略下 NCT 的性能

对于 II 型负载, CPIT 获得的总体性能总是小于 ICOUNT 策略下的总体性能, 特别的, 当百分比

很高或很低时, CPIT 相对于 ICOUNT 的总体性能损失更加严重. 主要原因是 ICOUNT 策略以优化总

体性能为目标,尤其是对 ILP 型负载的性能优化更加有效^[4].而在 CPIT 策略中,为了保证 CT 获得期望性能,往往需要强制性地使 CT 的取指优先级置为最高或最低.实际上,如果 CT 已经占用了很多的资源,这时把最高取指优先级赋给 CT 很有可能造成资源阻塞;而当 CT 能够利用 NCT 不需要的资源继续执行时,把 CT 的取指优先级降到最低会造成资源浪费.

对于 IM 型负载,随着目标百分比的提高,CPIT 相对于 ICOUNT 的总体性能也越来越高.这是因为,百分比越高,CT 的性能就越高.虽然 CT 性能的提高会导致 NCT 性能降低,但是由于 NCT 是 MEM 型程序而 CT 是 ILP 型程序,因此总体性能还是随着目标百分比的提高而提高.当目标百分比达到一定值时,CPIT 策略获得的总体性能甚至超过了 ICOUNT.不过对于不同大小的负载,CPIT 开始超过 ICOUNT 的百分比也不一样,对于 IM2, IM3 和 IM4,分别为 60%,50%和 40%.也就是说,负载中的线程数目越多,这个百分比越低.为了解释这个问题,我们首先分析使用 ICOUNT 策略时 CT 的性能.我们发现 IM4 中 CT 的性能最低,只有全速运行时性能的 32%,而对于 IM3 和 IM2,则分别达到了 42%和 51%.在 CPIT 策略下,当目标百分比超过这些值以后,CT 的性能就会超过 ICOUNT 策略下 CT 的性能.同样的,CT 性能的提高会导致 CPIT 策略下 NCT 的性能小于 ICOUNT 策略下 NCT 的性能(见图 5),但是由于 CT 对总体性能的贡献更大,从而使得 CPIT 策略获得了更高的总体性能.

对于 MI 型负载,可以得出和 IM 型负载相反的结论:随着目标百分比的提高,CPIT 相当于 ICOUNT 的总体性能越来越低.这是因为,当目标百分比很低时,CT 需要的资源很少.因此,ILP 型的 NCT 可以利用大多数共享资源获取很高的性能.从图 5 可以看出,与 ICOUNT 相比,CPIT 策略下 NCT 的性能在最好的情况下提高了 133.93%.

对于 MM 型负载,CPIT 相对于 ICOUNT 的总体性能与 II 型负载很相像.由于 CT 和 NCT 都是 MEM 型程序,因此,当 CT 的性能被强制提高时,NCT 的性能会以相同的幅度降低,反之也是如此.

总的来说,CPIT 策略在控制 CT 的性能时对 NCT 的性能和总体性能的影响并不大.与 ICOUNT 相比,总体性能平均降低不超过 3%,而 NCT 的性能平均只降低了 1.75%.

6 结 论

在 SMT 处理器中,同时运行的线程共享处理器资源,共享资源在线程之间的分配方式对处理器的总体性能和每个线程的性能都具有决定性的影响.当前,共享资源的分配主要由取指策略隐式决定.然而,已有的取指策略几乎都是针对如何提高总体性能的.

本文提出了一种新颖的取指策略 CPIT,用于控制 CT 的性能.CPIT 策略根据 CT 的当前执行进度在 PCT, ICOUNT 和 PNCT 策略之间切换,通过实时的调整 CT 的取指优先级来控制它的执行速度,从而控制它的性能,使其能够达到期望目标.模拟结果表明:

(1)对于模拟的所有负载,CPIT 策略的成功率超过 94%,也就是说对 94%以上的情况,CPIT 都能保证 CT 获得期望性能.对于失败的情形,CT 的平均性能偏差不超过 1.25%.

(2)CPIT 策略在控制 CT 性能时对 NCT 的性能和总体性能的影响并不大.与 ICOUNT 策略相比,NCT 的性能平均只降低了 1.75%,而总体性能的平均降低也不超过 3%.

参 考 文 献

- [1] Tullsen D, Eggers S, Levy H. Simultaneous multithreading: Maximizing on-chip parallelism//Proceedings of the Annual International Symposium on Computer Architecture, Santa Margherita Ligure, Italy, 1995: 392-403
- [2] Tullsen D, Eggers S et al. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor//Proceedings of the Annual International Symposium on Computer Architecture, PA, USA, 1996: 191-202
- [3] Eggers S J, Emer J et al. Simultaneous multithreading: A platform for next-generation processors. IEEE Micro, 1997, 17(5): 12-19
- [4] Tullsen D, Brown J. Handling long-latency loads in a simultaneous multithreaded processor//Proceedings of the Annual ACM/IEEE International Symposium on Microarchitecture, Texas, USA, 2001: 318-327
- [5] El-Moursy A, Albonesi D. Front-end policies for improved issue efficiency in SMT processors//Proceedings of the International Conference on High Performance Compute Architecture, California, USA, 2003: 31-42
- [6] Anantaraman A, Seth K et al. Virtual simple architecture

- (visa): Exceeding the complexity limit in safe real-time systems//Proceedings of the Annual International Symposium on Computer Architecture. California, USA, 2003: 350-361
- [7] Snively A, Tullsen D, Voelker G. Symbiotic Jobscheduling with priorities for a simultaneous multithreading processor//Proceedings of the International Conference on Measurement and Modeling of Computer Systems. California, USA, 2002: 66-76
- [8] Cazorla F, Knijnenburg P et al. Predictable performance in SMT processors//Proceedings of the ACM International Conference on Computing Frontiers. Ischia, Italy, 2004: 171-182
- [9] Cazorla F, Knijnenburg P et al. QoS for high-performance SMT processors in embedded systems. IEEE Micro, Special Issue on Embedded Systems, 2004, 24(4): 24-31
- [10] Cazorla F, Knijnenburg P et al. Architectural support for real-time task scheduling in SMT processors//Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems. California, USA, 2005: 166-176
- [11] Sherwood T, Calder B. Time varying behavior of programs. University of California, USA: Technical Report UCSDCS99-630, 1999
- [12] Tullsen D. Simulation and modeling of a simultaneous multithreading processor//Proceedings of the Annual Computer Measurement Group Conference. San Diego, CA, USA, 1996: 819-828
- [13] Sherwood T, Calder B et al. Basic block distribution analysis to find periodic behavior and simulation points in applications//Proceedings of the International Conference on Parallel Architectures and Compilation Techniques. Barcelona, Spain, 2001: 3-14



SUN Cai-Xia, born in 1979, Ph.D., lecturer. Her research interests include microprocessor computer architecture, multithreading, single-chip multi-processor, and VLSI design.

ZHANG Min-Xuan, born in 1954, professor, Ph.D. supervisor. His research interests include high-performance computer architecture, microprocessor design, low-power design and ASIC techniques.

Background

This paper focuses on fetch policies in Simultaneous Multithreading (SMT) processors and studies how to control the performance of a particular thread in SMT processors by fetch policies. Currently, fetch policies in SMT processors almost aim at how to optimize overall performance, and provide no control over how individual threads are executed.

In this paper, a novel fetch policy called CPIT (Controlling Performance of Individual Thread) is proposed to implicitly control the execution of a particular thread in SMT processors and how to implement CPIT is detailed also in this paper. According to the analysis and experiments, CPIT is proved to be practical and effective. For more than 94% of all cases measured, CPIT can control the execution and consequently achieve the desired performance for a given thread. For the failing cases, the average variance is within 1.25%. Furthermore, CPIT does not sacrifice overall performance of SMT processors severely. Compared to fetch policies orienting to-

wards performance maximization such as ICOUNT, the average degradation of overall performance is not more than 3% and the degradation of threads other than the given thread in performance is only 1.75%.

This work is supported by the National Natural Science Foundation of China under grant No. 60376018 and the National High Technology Research and Development Program (863 Program) of China under grant No. 2005AA11002. Both projects study the design of high-performance general processors and hope to make a contribution to processors made in our country. Their work mainly focuses on the microarchitecture, especially on multithreading and single chip multiprocessors. The research group has achieved much. Over one hundred papers were published and seven patents were granted. Over ten students had MS and four students had PhD with the support of these two projects.