

引用格式:张多利,陈楠,汪杨,等. 粗粒度多核系统任务级多线程调度研究[J]. 微电子学与计算机, 2020, 37(1): 46-52.
[ZHANG D L, CHEN N, WANG Y, et al. Research on task-level multi-thread scheduling for coarse-grained multi-core systems[J]. Microelectronics & Computer, 2020, 37(1): 46-52.]

粗粒度多核系统任务级多线程调度研究

张多利, 陈楠, 汪杨, 宋宇鲲

(合肥工业大学 电子科学与应用物理学院, 安徽 合肥 230009)

摘 要: 多核系统是当今处理器发展的主方向, 如何合理高效进行任务调度, 确保全部处理核心处于有效工作状态是当今多核系统研究的一个重要方向. 多核任务调度的关键难点在于发掘任务并行性, 为解决这一问题, 本文借鉴指令级多线程思想, 结合多核系统中任务的粗粒度特性, 提出了一种新型的粗粒度多线程多核体系结构, 建立了多线程取指策略、资源分配策略和线程切换机制, 同步完成了这一结构多线程调度器电路设计. 围绕此调度器构建了一个粗粒度多核计算平台, 并在 FPGA 芯片上进行硬件实现, 实验结果表明, 该设计方案相对于单线程使多核计算平台的任务并行度平均提高约 34.29%.

关键词: 多核技术; 多任务并行; 同时多线程; 粗粒度

中图分类号: TN47

文献标识码: A

文章编号: 1000—7180(2020)01—0046—07

Research on task-level multi-thread scheduling for coarse-grained multi-core systems

ZHANG Duo-li, CHEN Nan, WANG Yang, SONG Yu-kun

(School of Electronic Science and Applied Physics, Hefei University of Technology, Hefei 230009, China)

Abstract: Nowadays Multi-core system is the main direction of processor development. How to reasonably perform task scheduling and ensure that all processing cores are in effective working state is an important direction of multi-core system research. The key difficulty of multi-core task scheduling is to explore the parallelism of tasks. To solve this problem, this paper draws on the instruction-level multi-threading idea and combines the coarse-grained characteristics of tasks in multi-core systems to propose a new coarse-grained multi-threaded multi-core architecture. A multi-threaded fetching strategy, resource allocation strategy and thread switching mechanism are established, and the multi-threaded scheduler circuit design of this structure is completed. A coarse-grained multi-core computing platform is built around this scheduler, and implemented on the FPGA. The results show that the design scheme improves the task parallelism of the multi-core computing platform by about 34.29% on average compared with single-threaded.

Key words: multi-core technology; multitasking parallel; multithread; coarse-grain

1 引言

随着数字系统对 QoS 和集成电路制造工艺持续提升, 多核处理器已经成为处理器发展的主要方

向^[1-2]. 发挥这些芯片强大运算能力的前提是能够将计算划分为粗粒度任务, 并且通过数据通信和同步来协调这些任务的执行, 这个过程对于编程人员来说依然十分困难.

同时多线程技术可以看成在同一个处理器上虚拟多个逻辑处理器^[3],每个逻辑处理器上运行一个程序线程,但它们同时共享一套执行部件.同时多线程技术通过将其它线程的指令发往空闲功能单元,将线程级并行转换为指令级并行,有效提高处理器的指令吞吐率和功能单元的利用率^[4].多核系统可以借鉴多线程技术将任务发往空闲处理器单元,提高任务的并行度和处理器的利用率.

本文提出了一种粗粒度多线程多核系统体系结构(CMMA, Coarse-grained multi-threaded multicore architecture).CMMA 通过增加控制处理器(CP, control processor)和通用寄存器文件(URF)实现传统多核系统的多线程扩展. CP 负责调度任务分配,使用同时多线程技术提取任务之间的并行性.传统同时多线程技术是针对细粒度环境,且机器指令的输入输出有限.任务级并行的粗粒度特性和灵活性一方面限制了同时多线程技术的直接使用,但另一方面允许采用更简单的设计来实现这些技术.

2 相关工作

随着多核平台的不断普及,计算并行性研究从指令级扩展到任务级.传统指令级并行算法,比如记分牌和 Tomasulo 算法,是用多个功能单元挖掘 ILP 的有效方法.但是随着基于任务的编程模型的日益普及,研究重心转向通过任务级调度算法来实现并行.

任务级调度算法包括静态调度^[5-7]和动态调度^[8-10]算法,不过没有考虑到任务依赖性,这严重限制了任务级并行化.

任务级并行研究^[11-13]均建立在指令级并行技术基础上,指令并行技术发掘任务并行性有限,而多线程技术通过线程级并行充分发掘并行性,因此本文借鉴多线程技术充分发掘粗粒度多核平台的任务并行度.

3 粗粒度多核系统体系结构

如图 1 所示,粗粒度多核体系(CMMA)包含控制层和执行层两个层次.执行层是典型的共享存储多核处理器体系结构,由多个处理单元(PU)、互连网络和共享存储单元组成.处理单元(PU)可为通用处理器/数字信号处理器,或专用加速单元(IP 核),或任何其它类型的可编程单元. PU 主要执行粗粒度计算,本文称之为任务.

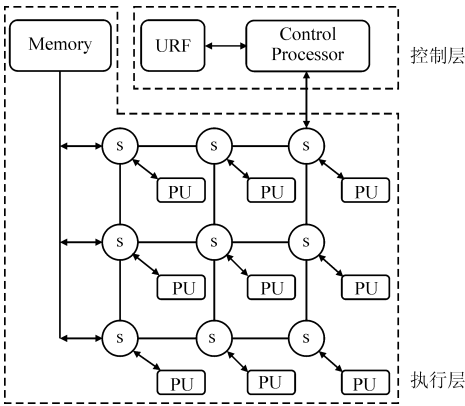


图 1 粗粒度多核架构示意图

控制层由控制处理器和通用寄存器文件组成.控制处理器获取任务指令并解码,然后将其调度发射到相应处理单元.任务指令用于指示任务的调用,使用通用寄存器作为其“输入”和“输出”映射表,即每个寄存器对应一块存储空间,运算单元根据寄存器编号去访问相应的存储空间的机制,解决了由于任务粒度较大无法直接使用寄存器的问题.

CMMA 的编程模型类似于一些粗粒度数据流编程模型,如 StarSs^[14], Intel RapidMind^[15] 和 OoOJava^[16].在本文编程模型中,任务指令指程序中的函数调用,任务指函数主体.系统程序由一组顺序任务函数和对应顶层顺序控制程序组成,这里每个任务函数代表一个任务.

CMMA 采用多线程编程,将处理器作为多个虚拟的处理器进行处理,并不需要专门对应用程序进行调整,并且针对单线程开发的应用代码不需要进行任何改动就可以移植到本文架构中.

图 2 给出一个简单的控制程序.函数调用表示任务指令,每个函数的参数都标记为 in 或 out 参数,以指示相应任务对该参数是读取还是写入,并维持任务之间的数据流向.

```
procedure thread 1
begin
A(out a)
B(in a, out b);
C(in a, in b, out c);
D(in b, out d);
E(in c, in d, out e);
end
procedure thread 2
begin
A(out a1)
B(in a1, in b, out b1);
C(in a1, in b1, out c1);
D(in b1, out d1);
E(in c1, in d1, out e1);
end
```

图 2 控制程序示意图

图 3 显示了上述程序的执行图,图中节点表示任务,实线表示任务间依赖关系.两个线程以重叠方式并行执行可获得近 2 倍的加速.

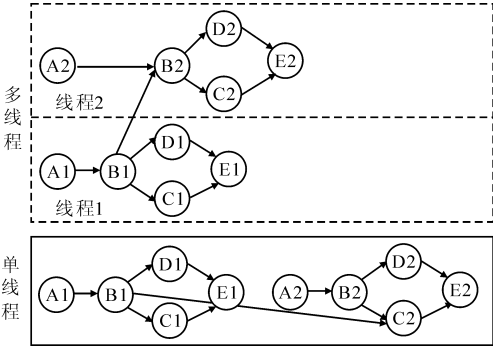


图 3 任务执行示意图

图 4 显示了执行上述程序处理器功能单元执行槽的表现状况,水平维度表示每个时钟周期中的执行槽的使用情况,垂直维度代表时间周期序列.空白框表示执行槽未使用,黑色和灰色框表示执行槽被不同的线程使用.由图可知,单线程处理器中,由于缺乏指令并行执行机制,功能单元的利用非常有限;而在多线程处理器中通过切换到另一个线程来隐藏长时间的停顿,减少了完全空闲的时钟周期数,提高了功能单元的利用率.

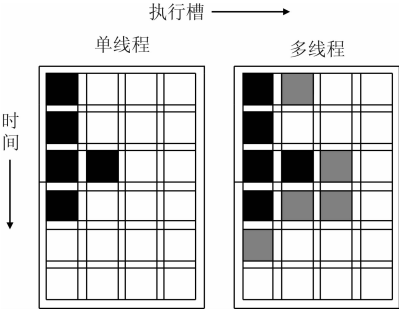


图 4 处理器功能单元执行槽表现状况

4 控制处理器微体系结构

4.1 概述

如图 5 所示为多线程控制处理器的整体结构,包含一个有序前端单元、一个乱序执行单元和一个线程管理单元.

线程管理单元包含仲裁模块、线程状态控制模块和线程保留栈.线程状态控制模块保存和控制线程的运行状态,仲裁模块根据各线程的运行状态进行仲裁,选择两个线程进行执行,并将相应的线程号送入前端单元执行.线程保留栈保存着线程切换后原线程的状态信息,用于线程恢复.

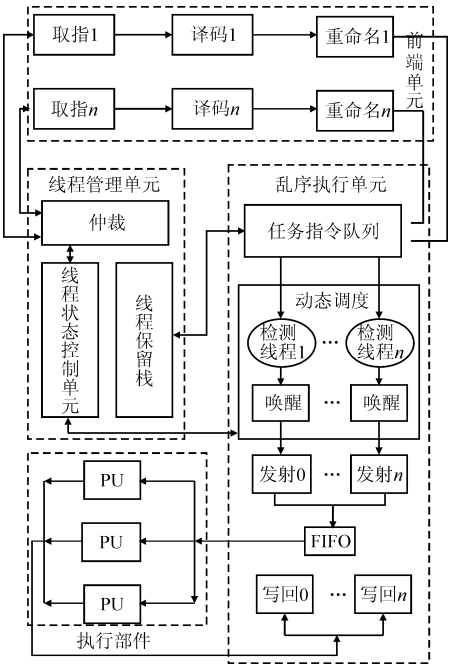


图 5 控制处理器架构示意图

前端单元从控制处理器的控制程序存储器中取出任务指令并解码,对任务指令的输入输出寄存器进行重命名,然后将任务指令发送给乱序执行单元.

乱序执行单元是控制处理器的执行部分,主要模块是任务队列、动态调度模块和发射模块.任务指令到达乱序执行单元后进入任务队列,等待其输入准备就绪.动态调度模块检测任务队列中的任务指令的输入是否准备就绪,一旦准备就绪,任务就会被唤醒并被转发到发射模块等待发射,同时任务队列中该任务指令清除.发射模块接收到就绪任务指令,为其分配一个空闲的执行单元 (PU),并将该任务指令对应的任务发送给分配的执行单元去执行.

本文多线程控制处理器采用粗粒度流水线,即取指、译码、重命名、调度、发射以流水线方式运行,每一级功能在多个周期完成.考虑到本文中任务的执行粒度通常在 K 个周期级,流水线消耗的周期相对于整体任务的执行周期可以忽略不计.因此使用粗粒度流水线可以简化设计,避免了在单周期完成复杂的组合逻辑,提高系统主频.

4.2 取指策略

由于每个线程处理指令的速率不同,并且存在对处理器共享资源的竞争,需设计合理取指策略,提高取指效率和质量.现有的多线程取指策略大体可分为 3 类:基于指令吞吐率、基于取指公平性和基于

资源分配.但这三类取指策略都是基于细粒度通用处理器的,不能直接应用于粗粒度系统.

本文考虑到任务指令的粗粒度特性,在基于指令吞吐率策略的基础上加入线程控制机制.如图6所示,本文策略主要分为两个部分:线程选择和取指过程.

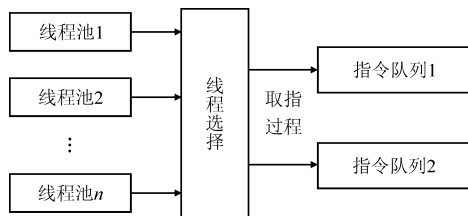


图6 取指策略示意图

(1)线程选择

取指部件选择线程个数和确定线程指令的操作.本文多线程控制处理器最多支持8个线程指令的处理,可同时对两个线程的指令进行操作.线程管理单元通过仲裁模块每次从8个线程中选择两个线程执行.

线程选择应优先选择指令执行速率最快的2个线程,本文以指令计分作为衡量线程指令执行速率的标准.评分公式如下:

$$\text{Score}_n = S_{\text{IO}} + S_{\text{MEM}} + S_{\text{CAL}} \quad (1)$$

式中, S_{IO} 、 S_{MEM} 和 S_{CAL} 分别表示每次指令队列发射的输入输出指令、访存指令以及计算指令的指令数. Score_n 反映线程任务指令的执行程度,其值越大表示单位评估周期内发射的指令越多,即当前线程的任务粒度相对较大.因此以各线程的评分结果 Score_n 来决定取指的优先级, Score_n 越大优先级越高.

(2)取指过程

取指过程操作如下:在线程切换时清空指令队列,取指单元从任务指令Cache读取指令填满指令队列,然后队列中每发射出一条任务指令,取指单元会自动读取一条新指令填充队列.

选择这种取指操作是因为任务指令的粗粒度特性,首先一条任务指令占据8个指令字,如果每次读取多条任务指令可能引起任务指令Cache失效导致线程无法继续取指,所以每发射一条任务指令后进行一次取指;其次任务指令执行周期长,每个线程的取指频率较低,指令带宽压力较小.

4.3 资源分配策略

在多核系统中,承载任务指令操作的具体操作的单元数量多、种类丰富,当多个线程的指令同时分

发至同一个运算单元时,这些指令间存在竞争.而各个线程的指令执行效率不尽相同,应尽可能将更多的资源分配给指令执行效率高的线程.

本文利用指令主动请求资源的方式对共享资源统一管理,指令经取指进入流水线后,即分配确定的系统资源,因此本文分配策略以指令队列为核心,通过调整每个线程指令队列占用的队列数量间接控制每个线程占用处理器资源的使用.

图7所示为指令队列的划分方式,处理器有两个硬件线程,每个线程都有一个深度为32的指令队列.两个线程对应的指令队列分成4部分(Block0、Block1、Block2、Block3),其中Block0和Block2分别作为线程A和线程B的静态队列,存储相应线程的任务指令,而Block1和Block3采用动态分配方法,根据线程评分结果分配给相应的线程.

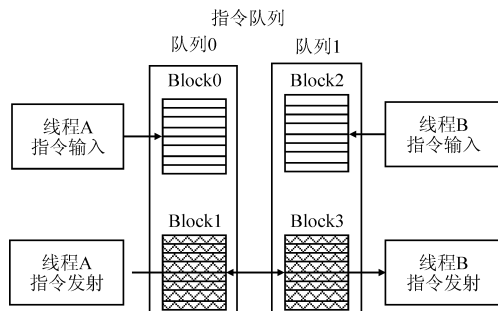


图7 指令队列划分

从上节可知 Score_n 反映了线程任务指令的执行速度,而 Score_A 和 Score_B 的比值反映了两个线程执行的快慢.本文根据 Score_A 和 Score_B 的比值分配Block1和Block3,分配方式如下:

- (1) $\frac{\text{Score}_A}{\text{Score}_B} \geq 2$:表示线程A执行速度远大于线程B,将Block1和Block3分配给线程A;
- (2) $\frac{\text{Score}_A}{\text{Score}_B} \leq \frac{1}{2}$:表示线程A执行速度远远小于线程B,将Block1和Block3分配给线程B;
- (3) $\frac{1}{2} \leq \frac{\text{Score}_A}{\text{Score}_B} \leq 2$:表示线程和线程B执行速度相当,将Block1分配给线程A,Block3分配给线程B.

通过这种动态分配使得每个线程都能分配到足够的指令和资源,并且为指令执行速率快的线程分配更多的资源.

4.4 线程切换机制

控制处理器在遇到L2 Cache缺失、分支预测错误、访存时间较长、缺少资源等情况时线程会处于阻

塞停滞状态,需要进行线程切换,通过切换到另一个线程来隐藏长时间的停顿,减少完全空闲的时钟周期数.在线程切换时线程状态的现场保护和共享资源的同步释放至关重要,当其它线程执行完成或阻塞切换回到该线程时,可以快速恢复到切换前的状态.

传统细粒度系统会为各个线程创建一个私有的存储空间,保存寄存器映射关系以及指令队列等信息.在本文粗粒度多核系统中若为每个线程创建私有存储,由于任务指令执行周期长,线程会占用较长时间,一旦保存所有线程的线程状态信息会长时间占用存储空间,而且一些共享资源若不释放也会产生极大的浪费,而在细粒度系统由于线程频繁切换不会导致资源长期占用.

本文从资源和效率平衡目标出发,在线程切换时以少量的存储空间保护现场,并及时释放指令中占有的资源.图 8 描述了线程切换时对线程的保护机制.在进行切换时,将指令中的任务序号和当前 PC 指针记录在对应线程编号的记录表中,不需要保存指令队列,节省存储空间,并且释放输出物理寄存器,将映射关系保存在寄存器保存表.线程恢复时,根据对应的记录表记录的 PC 地址重新取指,同时将任务号恢复到当前指令中,然后根据寄存器保存表恢复寄存器映射关系,至此恢复完成,等待调度单元重新检测指令执行状态.

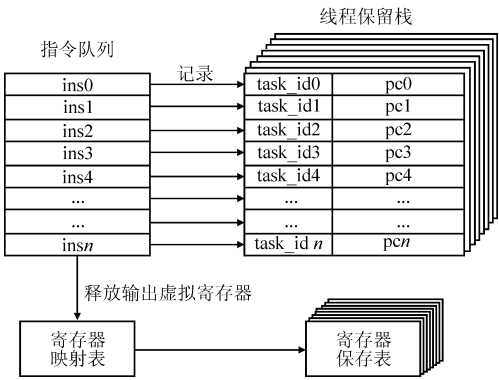


图 8 线程切换保护机制

5 实验评估

论文根据所提出多线程体系结构设计了一个多核系统,包含多线程控制处理器、32 个 PU 和一个存储子系统,并在 Xilinx Virtex-7 XC7V2000T FPGA 芯片上实现了系统硬件原型. 100M 工作频率下,控制处理器单元的资源消耗如表 1 所示,可以看出多线程控制处理器单元使用的资源较少.

表 1 控制处理器单元资源消耗

Resource	FF	LUT	BRAM
消耗/个	12 945	11 585	8.5
消耗占比	0.23	0.51	0.67

为了评估本文提出的多线程控制处理器的并行调度能力,下文设计了不同特征任务的加载实验,并与单线程架构加载结果进行对比.

5.1 测试用例和测试过程

本文采用任务指令并行度(简称 T_{ILP})作为同时多线程架构的性能指标. T_{ILP} 的表达式如式(2)所示:

$$T_{ILP} = \frac{T_{theory}}{T_{actual}} \tag{2}$$

式(2)中 T_{theory} 指未经过并行处理的任务指令理论执行时间, T_{actual} 表示实际执行时间,任务指令并行度表示任务指令的并行度,反映了系统的并行处理能力.

在开始实验之前,本文选取了 4 个测试任务(task1、task2、task3、task4),构造了具有不同相关性的任务序列,用于模拟多任务环境,这四个测试任务由不同的 PU 单元执行完成.任务相关性是相对于任务之间的依赖性而言,其定义如定义 1 所示.比如一个任务序列的相关性为 100%,即这个指令序列中的所有指令都与前一条指令具有数据依赖性,所有指令必须串行执行.

定义 1 任务相关性,是指具有真数据相关的任务在所有任务指令中的比例.

本文构造不同任务相关度的任务序列进行实验,测试不同任务相关度下的性能差异.如图 9 所示,分别显示了相关度为 25%,50%,75%,100%的任务序列构造方法,图中实线箭头相连的任务具有数据相关性,无实线相连的任务没有相关性.

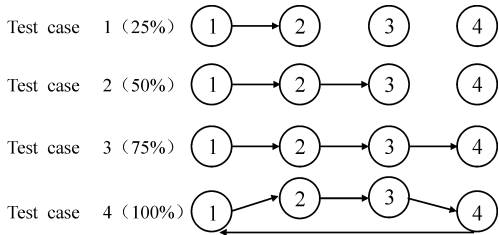


图 9 测试用例

通过调整四个测试任务的输入寄存器和输出寄存器,使其构成相应的任务数据相关,然后将这四个测试任务作为循环体循环若干次数,构造不同长度的任务序列.例如测试用例 2 中的 4 个测试任务有

两个数据相关,再循环 100 次,那么 400 个任务中有 200 个数据相关,最终得到长度为 400 任务相关性为 50% 的任务序列.也可以使用 3 个测试任务、5 个测试任务甚至更多作为循环体构造不同任务相关性的任务序列.

5.2 任务粒度的影响

为了测试任务的数据粒度对多线程的影响,使用上述任务相关性为 0 和 100% 的任务序列作为实验对象并改变任务指令的输入数据粒度进行实验.每组实验使用 4 线程编程,每个线程一组长度为 100 的任务序列,线程之间没有数据相关性,两组实验结果如图 10 所示.

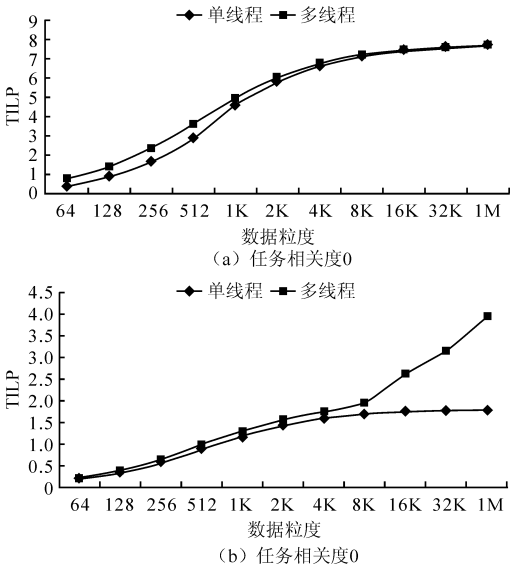


图 10 任务粒度实验

图 10(a)为任务相关性为 0 的实验结果,从图中可看出在数据粒度小于 2 k 时多线程方案对系统的加速效果优于单线程,当数据粒度大于 2 k 时,多线程和单线程方案相差不大都趋近于 8. 这是因为当数据粒度较小时,任务执行时间较短,指令发射的时间间隔对总体任务执行时间影响很大,而多线程拥有两个发射槽,可以实现指令的同时双发射,因此对于多线程对粒度较小的无关任务有明显的加速效果.然而当数据粒度大于 1 k 时,任务的执行时间较长,计算资源长时间被占用,当所有的资源处于被占用的状态时,多线程相对于单线程没有明显的优势,因此两者的任务并行度相差无几.

图 10(b)为任务相关性为 100% 的实验结果.对于单线程,调度窗口深度设为 16,当第一组指令的第 10 条指令执行完后,第二组指令开始进入调度窗口,此时存在两条不相关指令,可以同时发射,之后

便一直是这样,因此总体任务的 T_{ILP} 有所上升,随着数据粒度的增加,这种增长趋势会越来越缓,最终趋向于 1.82. 在多线程模式,在数据粒度较小时,两个线程只能同时处理两条指令,因此 T_{ILP} 随着粒度的增加趋近于 2. 而本文多线程在线程阻塞时会进行线程切换,当某一线程占用超过阈值时间会触发线程切换,本实验阈值设为 8 k 周期.当数据粒度超过 8 k 时,运算周期也超过了 8 k,此时会触发线程切换,切换到另一线程去执行,使得 T_{ILP} 随着粒度的增加显著增加.

5.3 任务相关性的影响

从上节可以看出,任务的相关性影响到多线程的并行调度,为了分析任务相关性的影响,本实验构造了不同任务相关性的任务序列进行实验,任务序列长度为 100,所有任务的数据粒度为 16 k,实验结果如图 11 所示.

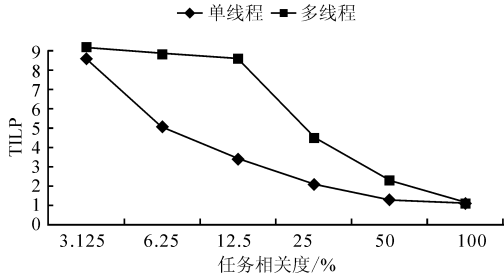


图 11 任务相关性实验

从图 11 可以看出,对于不同的任务相关性多线程模式的 T_{ILP} 普遍高于单线程,采用多线程技术可以将 T_{ILP} 平均提高约 34.29%.

图 11 中随着任务相关性的增加,多线程与单线程的 T_{ILP} 差值先增加后减小,最后趋于相同.产生这种现象的原因是当任务相关性较小时,每个线程内的调度窗口容纳的任务指令多,可同时发射多个无关指令,单线程和多线程的差距不是很明显;而随着任务相关性的增加,调度窗口内可同时发射的指令数减少,单线程调度能力有限,而多线程可通过线程切换执行多个线程的指令,大大提升了指令并行度.而随着任务相关性的继续增加,任务无法并行,因此多线程和单线程结果一样.

6 结束语

本论文为粗粒度多核系统提出了一种任务级多线程架构,并设计实现了一款支持同时多线程的控制处理器.针对粗粒度多核系统任务特点提出了一种适用于目标系统的多线程取指策略和资源分配策

略,提高了控制处理器的任务发射效率,提升了系统的平均任务并行度.相较于单线程的对照设计,本文多线程调度方案使得粗粒度多核系统的计算性能得到了极大发挥,充分地挖掘了系统的并行计算潜力.

参考文献:

- [1] GEER D. Industry trends: chip makers turn to multicore processors[J]. Computer, 2005, 38(5):11-13.
- [2] KALLA R, SINHARROY B, TENDLER J M. IBM Power5 chip: a dual-core multithreaded processor[J]. Micro IEEE, 2004, 24(2):40-47.
- [3] 沈钰,孙义和. 一种支持同时多线程的 VLIW DSP 架构[J]. 电子学报, 2010, 38(2):352-358.
- [4] 李乐,李涛. 同时多线程处理器的指令调度器设计[J]. 微电子学与计算机, 2016, 33(6):27-31.
- [5] GREWE D, O'BOYLE M F P. A static task partitioning approach for heterogeneous systems using OpenCL[M]. Compiler Construction. Berlin, Germany: Springer-Verlag, 2011, :286-305.
- [6] 娄建峰,高岳林,李飞,等. 基于改进粒子群算法的云计算任务调度算法[J]. 微电子学与计算机, 2016, 33(8):112-116.
- [7] 张海玉. 基于改进蚁群算法的云计算任务调度研究[J]. 微电子学与计算机, 2016, 33(9):110-113.
- [8] ARNOLD O, FETTWEIS G. On the impact of dynamic task scheduling in heterogeneous MPSoCs [C]// Proc. Int. Conf. Embedded Comput. Syst. (SAMOS). Greece, 2011:17-24.
- [9] 吕鹏伟,刘从新,沈绪榜. 一种面向嵌入式多核系统的任务调度方法[J]. 微电子学与计算机, 2017, 34(4):1-7.
- [10] LUK C K, HONG S, KIM H. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping [C]//Proc. 42nd Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO). [S. l.]. 2009:45-55.
- [11] DIAMOS G F, YALAMANCHILI S. Harmony: An execution model and runtime for heterogeneous many core systems[C]// Proc. 17th Int. Symp. High Perform. Distrib. Comput. Miami, Florida USA, 2008:197-200.
- [12] CAPALIJA D, ABDELRAHMAN T S. Microarchitecture of a coarse-grain out-of-order superscalar processor[J]. IEEE Transactions on Parallel and Distributed Systems, 2013, 24(2):392-405.
- [13] GUO Q, LI X, WANG C, et al. Evaluation and tradeoffs for out-of-order execution on reconfigurable heterogeneous MPSoC [J]. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 2015, 24(1):1-1.
- [14] PEREZ J. A dependency-aware task-based programming environment for multi-core architectures [C]// Proc. IEEE Int'l Conf. Cluster Computing. [S. l.]. 2008.
- [15] MCCOOL M D. Data-Parallel Programming on the Cell BE and the GPU Using the RapidMind Development Platform[C]//Proc. GSPx Multicore Applications Conf. , Santa Clara, 2006.
- [16] JENISTA J C. OoJava: an out-of-order approach to parallel programming [C]//Proc. Second USENIX Conf. Hot Topics in Parallelism (HotPar '10). Berkeley, CA, 2010.

作者简介:

张多利 男, (1976-), 博士, 研究员, 硕士生导师. 研究方向为多核处理器体系结构与设计方法.

陈楠 男, (1993-), 硕士研究生. 研究方向为集成电路设计与测试. E-mail: 15256267553@163.com.

汪杨 男, (1996-), 硕士研究生. 研究方向为集成电路设计与测试.

宋宇鲲 男, (1975-), 博士, 副研究员. 研究方向为面向数据密集与计算机密集应用的 SoC/MPSoC 体系结构与实现.