中国科学 E 辑：信息科学　2008 年 第 38 卷 第 4 期：579 ~ 591
www.scichina.com　info.scichina.com
《中国科学》杂志社
SCIENCE IN CHINA PRESS

支持循环自动流水线的粗粒度可重构阵列体系结构

窦勇，邬贵明*，徐进辉，周兴铭

国防科学技术大学计算机学院，长沙 410073
* 联系人, E-mail: wuguiming@nudt.edu.cn

收稿日期: 2007-03-05; 接受日期: 2007-05-09
国家自然科学基金(批准号: 60633050)资助项目

摘要　可重构计算兼顾定制计算的高效性与通用计算的灵活性，是人们正在探索的一类新型计算机体系结构. 文中介绍一种粗粒度可重构多核协处理器体系结构，用于加速计算密集或数据密集类算法. 在该结构中采用具有猜测执行机制的循环自动流水线技术，将流水线并行执行和数据驱动执行紧密结合. 采用以循环程序为单位的固定指令多数据流执行模式，在循环执行期间，每个单元固定执行一种指令，配置信息量减少，支持快速动态重构. 设计了多种特殊数据传递指令，如非平衡数据复制指令、数据选择指令等，充分利用循环迭代内部、循环迭代间的数据相关，开发数据重用性，减少对存储器带宽的需求. 以大容量 FPGA 开发板为基础，构建了 8×8 规模的可重构阵列实验环境. 多个计算密集型程序的实际运行结果表明，该体系结构在流水线吞吐率方面可以达到每拍 1 个循环迭代，访问存储器的请求数仅为典型 RISC 处理器的 3%，并且具有单拍切换配置现场的能力.

关键词
可重构计算
循环流水线
数据驱动
寄存器提升

0.1　概述

可重构计算的发展目标是具有定制计算的高效性，同时兼顾通用计算的灵活性. 面向特定应用的芯片设计(ASIC)，是实现定制计算的典型方法，设计人员用硬件描述语言或逻辑图将算法直接映射到硬件电路上，其设计能够开发算法的多种并行性和存储优化方法，具有很高的计算效率，但是由于算法与所设计的硬件结构密切结合，定制计算适应多种算法的灵活性差. 另一方面，通用微处理器以顺序执行机器指令的方式完成计算任务，采用高级语言编程，具有适应多种算法的灵活性，但是计算效率没有定制计算高.

可重构处理器由处理单元阵列、互连网络和配置存储器三大部分组成. 基本工作原理是将对应用算法的描述存放在配置存储器中, 该配置决定了每个处理单元的功能和互连网络的数据交换方式, 类似定制计算中算法到硬件电路的映射. 通过改变配置存储器的内容, 重新构造处理单元和互连网络, 可重构处理器能够处理各种应用算法. 可重构处理器定位于与通用微处理器配合作为算法加速协处理器使用. 随着大规模集成电路芯片容量的持续增加, 单芯片多核处理器体系结构已经成为未来发展方向, 可重构处理器可与通用微处理器构成异构多核体系结构. 通用微处理器擅长处理复杂控制型应用, 与之配合的可重构处理器处理计算密集型或数据密集型应用, 能够发挥各自的优势.

本文介绍了一种粗粒度可重构协处理器体系结构 LEAP(loop engine on array processors), 用于加速计算密集或数据密集类算法. 在该结构中采用了循环自动流水线技术, 将流水线并行执行和数据驱动执行紧密结合. 利用循环程序中计算任务在处理阵列上的时间重叠, 循环迭代之间形成流水线并行, 存在数据相关的语句之间构成运算流水线链接. 通过对循环程序的流水线并行, 提高可重构处理器的计算吞吐率, 理想目标是每时钟节拍完成一个循环迭代. 通过数据驱动机制将集中式循环流水线调度策略分解为每个处理单元上的分布式控制模式, 对处理单元的控制仅与数据到达的局部状态相关, 实现了循环程序在可重构阵列上的自主执行.

与国内外已有相关工作相比, 本文所提出 LEAP 体系结构的主要特点是:

● 具有猜测执行能力的循环自动流水线技术. 利用数据驱动执行, 采用分布式循环控制, 预先发射循环迭代, 实现了循环片猜测执行, 通过控制写回操作, 在循环遇到不确定性跳转指令(Break)时, 仍然能够保证循环正确执行.

● 支持快速动态重构. 在循环执行期间, 每个单元固定执行一种指令, 而且单元之间的连接关系固定, 配置信息量大大减少(每个计算单元仅需 32 位); 配置信息分布存储在每个单元的多层配置存储体中, 通过配置指令可以实现循环之间配置信息的自动切换.

● 设计了特殊指令充分利用数据依赖关系开发数据重用性. 设计了非平衡数据复制指令, 结合 FIFO 数据队列, 利用循环迭代间的输入数据重用; 设计数据选择指令, 利用循环迭代间的写后读数据重用; 设计了数据直传通道, 利用了循环迭代内的写后读数据重用.

0.2 相关工作

粗粒度可重构体系结构一直是可重构体系结构的研究热点. 从支撑体系结构的计算模型来分析, 粗粒度可重构体系结构可以分为以下几类:

■ 动态流水线执行

Cardoso[1]提出了循环流水线自动执行的编译方法, 程序执行的硬件平台是XPP[2]. 我们与之主要的不同在于, 提出了循环流水线猜测执行机制和支持多重For循环的分布式控制状态机, 提高了循环流水线的吞吐率. 另一方面, 我们设计了支持数据重用的非平衡数据复制指令, 有利于输入数据的重用.

■ VLIW 中的软件流水线

在软件流水线方面, Garp体系结构 [3~5]利用处理单元阵列实现可重构功能部件, 将循环体的操作组合成为一个宏操作在可重构硬件上实现, 相当于硬件实现了一个宏指令, 再按照模调度 [6]策略在编译阶段组织循环程序的软件流水线. 程序在执行时, 宏指令被发射到可重构单元阵列上执行. LEAP实现了在运行时硬件自动控制的循环流水线执行; 而软件流水线属于精确的静态调度方法, 该调度方法受限于实际程序执行时各种不确定性事件, 例如, cache失效或者存储体冲突导致的不确定性延迟.

■ SIMD

MorphoSys[7]主控制器与可重构阵列紧密耦合, 每个可重构单元通过二维网格连接, 可通过软件配置单元的计算功能. 多个单元按照单指令同时处理多个数据的方式工作; 多个处理单元通过可配置网络, 实现多种数据交换模式.

与LEAP相近的另一类体系结构是数据流体系结构. 数据驱动是数据流体系结构的核心思想, 纵观数据流体系结构的发展, 可以分为静态数据流体系结构 [8]、动态数据流体系结构 [9]和混合型数据流体系结构 [10]. 静态与动态的主要区别是数据通道能够容纳token的数量, 如果每个通道只能存储一个token为静态数据流, 否则为动态数据流. 数据通道容纳多个token表示循环的多个迭代可以同时并行执行, 否则只能按照流水线顺序执行. 传统的数据流体系结构中, 每个处理机执行一个数据流图, 相当于一段指令, 指令执行的结果token被送入指令匹配部件, 从中选择数据就绪的后续指令发射, 因此token在指令序列中的匹配操作需要相联存储器、Hash表存储器或特殊的指令存储器, 这种结构虽然能够达到数据驱动执行的目的, 但是相联存储的硬件开销大, 数据相关指令的延迟长, 是影响数据流体系结构发展的重要原因. 后续研究工作针对这种瓶颈问题进行了一系列改进, 包括线程化数据流(threaded dataflow)、粗粒度数据流和控制流与数据流混合型体系结构等等, 但是在多指令序列中匹配token的原理仍然存在.

我们在采用数据驱动机制时, 提出了以循环为单位每个 PE 固定执行一条指令的执行方法, 即固定指令多数据流执行 FIMD(fixed instruction with multiple dataflow), 避免了 token 与指令匹配的瓶颈问题, 同时具有固定网络连接关系和固定单元配置等优点, 便于简化处理单元设计, 提高循环流水线吞吐率.

1 虚拟指令与循环程序的 Kahn 进程网络描述

本文用Kahn进程 [11]定义了 8 条虚拟指令的执行语义(详见附录). 该虚拟指令集包括了描述循环程序执行的基本操作. 循环启动指令(G)向其他指令发出启动命令. 循环结束指令(E)根据输入的判断条件, 发出循环结束标志, 用于完成循环退出. LD指令描述了读存储器操作. ST指令描述了写存储器操作. MM指令描述了数据存储器. Compute计算指令描述了数据驱动形式的计算, 其结果可能是数值, 也可能是Boole值, 用于条件判断. 数据选择指令(J)根据输入的Boole判断条件, 从输入中选择数据传递到输出端. 数据多路复制指令(F)将输入数据传递到

多个输出端. 各个虚拟指令的接口描述如下:
Instruction F (input channel I; output channel L, R)//Data fork;
Instruction J (input channel L, R, P; output channel O)//Data join;
Instruction LD (input channel: Address-in, EXE-token, Return-data; output channel: address-out);
Instruction ST (input channel: Address-in, EXE-token, Return-data; output channel: address-out, Result-mm);
Instruction G (input signal: Loop-start; input channel: Loop-end; Output signal Loop-over; output channel: EXE-token);
Instruction E (input channel: Loop-check; output channel: Loop-end);
Instruction Compute (input channel: DataL, DataR; output channel: Result; function f());
Instruction MM (input channel: LD-Address, ST-Address, ST-Result; output channel: Reture-data).

在描述循环程序过程中, 用进程结点描述虚拟指令的实例化, 用边描述连接进程结点间的互连通道, 其构成的 Kahn 进程网络描述了循环程序按照数据流动执行的语义. 如图 1(a)表示了一个用高级语言描述的循环程序以及转换后的伪汇编代码, 最后转换为用虚拟指令描述的 Kahn 进程网络, 如图1(b)所示. 其中边(1)表示数据写后读相关, 边(2)和(3)表示数据复制, 边(4)表示测试结果, 边(5)表示循环启动, 边(6)表示写入控制, 边(7)表示循环结束.

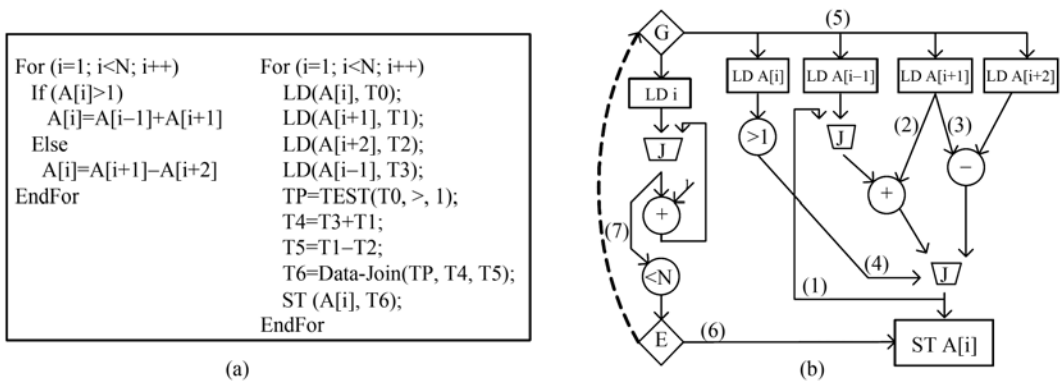

图 1

(a)高级语言描述的例子程序和伪汇编代码; (b)用虚拟指令表示的数据流图

虚拟指令不考虑实际的物理实现方法、执行效率和资源约束, 仅用以表达程序的语义. 一种虚拟指令可以对应有多种体系结构实现方法. 基于上述虚拟指令集, 我们提出了 LEAP 体系结构, 设计了实现虚拟指令的优化方法, 采用循环自动流水线猜测执行和存储访问优化技术, 提高循环程序在可重构体系结构上的执行效率.

2 粗粒度可重构体系结构 LEAP 设计

LEAP 体系结构的设计思想是, 假设数据已经存储在数据存储器中, 处理单元完成基本的虚拟指令功能, 通过互连网络将多个处理单元连接起来形成阵列. 将 Kahn 进程网络描述的算法映射到处理单元阵列和互连网络上, 虚拟指令映射到具体的物理处理单元, 虚拟指令之间的连接关系映射到互连网络的路由通道. 当注入启动信号后, 每个处理单元独立工作, 按照数

据驱动的方式进行计算; 当循环结束时, 发出循环结束信号, 等待下一次启动.

如图 2 所示, 可重构处理单元阵列 LEAP 体系结构由处理单元阵列、互连网络和接口控制器三部分组成. 处理单元分为两类: 用于循环控制和存储访问控制的 MPE 和用于计算任务的 CPE. 所有处理单元都和路由器相连, 路由器之间又相互连接构成一定拓扑结构的互连网络. 接口控制器是连接主控处理器核、外部主存储器和 LEAP 可重构阵列处理器的桥梁, 用于向 LEAP 阵列加载配置信息、以 DMA 方式在外部主存储器和内部数据存储器之间加载数据或读取结果或是主控制处理器向 LEAP 发送启动命令及查询运行状态.

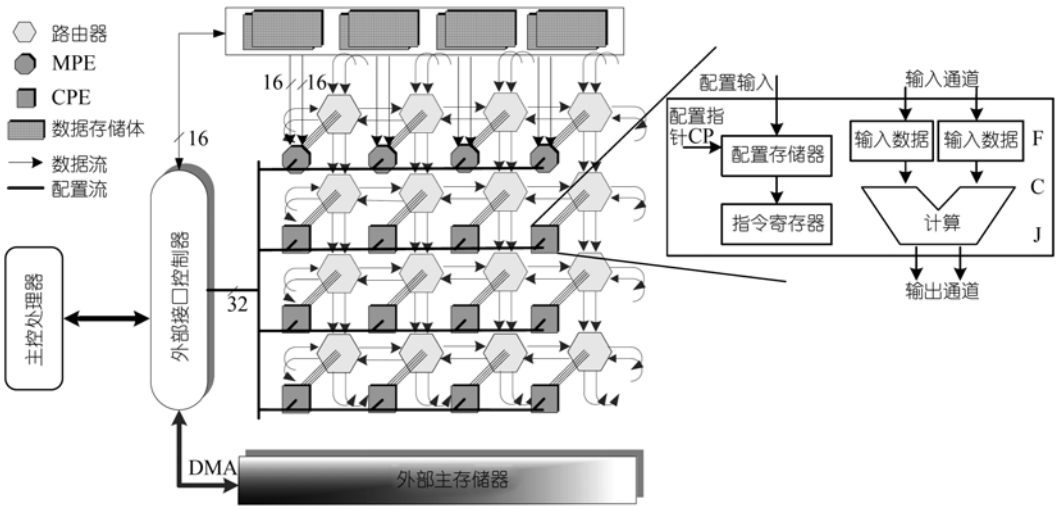

图 2　LEAP 可重构处理单元阵列体系结构与 CPE 结构

根据虚拟指令的功能和相互关联的特点进行模块化设计, 保证 LEAP 体系结构可以根据硬件资源情形灵活配置, 同时具有结构可扩展性. 存储指令 LD/ST 与数据存储器 MM 紧密关联, 负责循环控制的 G 指令与循环结束的 E 指令均与 LD/ST 紧密关联, 所以将它们集中在 MPE 中实现, 结构如图 3 所示, 多个 MPE 模块完成循环中的数据存取和循环控制. MPE 中间部分的地址队列、地址生成、取数控制和数据队列完成 LD 指令的功能. 在循环控制部件 G 的驱动下, 从地址队列中取地址或直接通过地址生成部件计算得到地址, 向数据存储器发出读数据请求, 返回数据写入数据队列中, 等待发送到 CPE 中. MPE 右侧部分的地址生成、存数控制和数据队列完成 ST 指令功能, 在循环结束部件 E 的控制下, 向数据存储器发出写请求. 通过对 LD/ST 指令寄存器的设置, MPE 支持多种数组访问模式. 首先是常用的三维线性下标数组访问, 例如 A[2i+1][j–1][k+5], 地址生成部件可以直接从循环控制部件中得到循环下标值, 按照地址计算公式 C1*i+C2*j+C3*k+C4 模式产生地址. 其次可以支持复杂数据访问模式, 此时地址由多个 CPE 计算得到, 通过输入通道, 送入地址队列. 还可以支持间址访问, 例如 A[B[i+1]], 此时用一个 MPE 取 B[i+1]的值, 然后送入 A 的地址队列. 有关 G 和 E 指令的实现方法结合循环自动流水线的控制方法在 3.1 小节介绍.

同样根据模块化的要求, CPE 实现数据流动过程中的计算和选择功能, 主要实现了 C 和 F 及 J 虚拟指令. CPE 的结构如图 2 所示, CPE 支持 2 操作数计算模式(T=TL op TR), 配置信息确定了当前执行的指令功能. 支持 F 指令从输入通道广播数据到 2 个输出通道, 多个 1-2 广播指令构成广播树, 支持 1 到多数据广播. CPE 支持分支数据选择指令 J(T=P?(TL: TR)), 该指令的语义是根据 P 的 Boole 值从两个输入中选择一个输出, 当 P 为真, 选 TL; 否则选 TR, P 是关系运算类指令执行的测试结果.

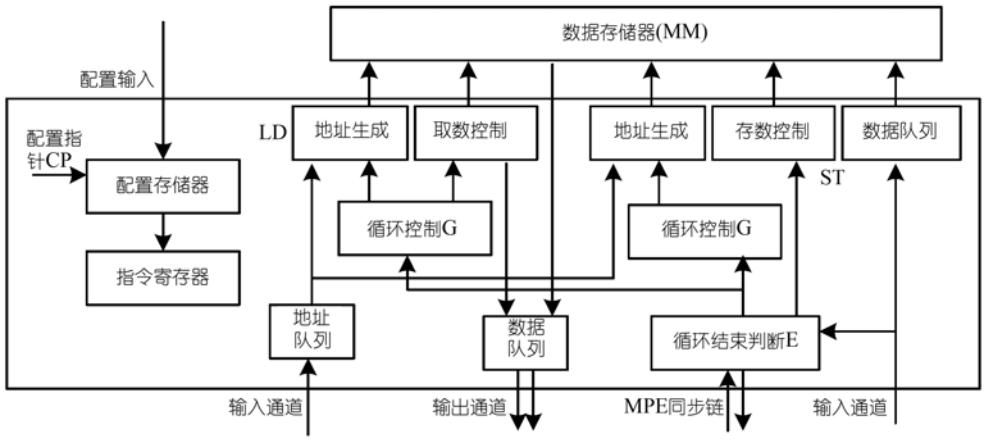

图 3　MPE 结构框图

互联网络通过路由器将各个处理单元连接在一起. 根据路由切换机制, 路由器可以分为静态连接和动态连接两大类. 比如在 FPGA 芯片中, 静态连接方法将逻辑块通过连线矩阵连接, 通过配置信息确定两条连接线的交叉点是否连通. 采用动态连接方法, 路由器的输入输出没有固定连接关系, 路由器根据报文目的结点号, 按照路由算法动态选择报文在路由器中通过. 我们以静态互连方法为基础, 设计了带缓存的可配置静态路由器. 配置存储器的每位单元控制输入和输出通道的连接点, 每条通道包括数据信号和数据有效信号以及一条前方状态反馈信号 ACK. 在每条输出通道出口处用寄存器将长线分段隔离, 以减小传输延迟. 循环程序映射到 LEAP 体系结构, 在执行过程中具有固定的流水线连接关系, 可以采用配置存储器控制交叉点的连接关系, 配置存储器内容随处理单元的配置信息切换. 该方法与动态连接相比, 不需要缓存数据报文, 不需要集中式报文仲裁和复杂的交叉开关, 仅需连接线和少量配置存储器, 因而非常有利于提高电路频率, 降低硬件复杂度.

配置信息确定了处理单元的功能和互连网络的连接关系, 配置存储器分布于每个 PE 中. 每个 PE 有一套配置存储器和配置控制部件, 用于配置信息的加载和存储. MPE 配置存储器位宽在逻辑上有 224 位, 物理上由 7 个 32 位宽存储体组成, 主要用于存储多重循环的控制参数、读写地址表达式参数. CPE 配置存储器位宽为 32 位, 用于指定 CPE 的操作类型和路由. 配置存储器的深度有 16 层, 有一个配置指针标识当前在用的配置层号. 主控处理器通过配置指针有选择地切换配置层号, 可以支持多个循环同时在阵列上执行, 也可以支持多个循环以不同

配置层的方式顺序在阵列上执行，以及部分 PE 接收配置信息，另一部分 PE 重叠计算. 有两种配置切换方式，一种是被动配置执行，由主控处理器通过设置当前配置指针指向对应的配置层号；另一种是自动配置执行，在当前配置执行结束后，配置控制逻辑自动更新配置指针，新的配置立即生效，开始新的循环执行，无需等待外部触发.

LEAP 体系结构通过模块化设计，以处理单元为单位，采用多种可配置指令，实现了 Kahn 进程网络中虚拟指令的功能. 但是，按照 Kahn 进程网络描述的语义，循环控制指令与循环结束指令需要通过传递信息保持同步，限制了指令流出的吞吐率. 在下一节，介绍猜测执行和循环控制部件的优化方法，和为了减少对存储器带宽的需求，LEAP 体系结构针对数据重用所做的指令优化.

3　可重构处理单元的优化设计方法

3.1　支持猜测执行的循环自动流水线

循环自动流水线的设计思想是将循环映射到处理单元阵列之后，通过配置循环控制变量的初值、终值和步进值，循环程序的执行不需要外部控制，处理单元之间构成流水线链接，能够自动完成循环迭代在流水线上的调度和执行. 循环自动流水线的目标是每拍调度一个循环迭代执行.

在调度循环迭代的过程中，更新循环控制变量的间隔是影响流水线吞吐率的重要因素之一. 图 4 所示, Budiu[12]提出的“Dataflow Software Pipeline”数据流图和循环调度方法，其中，更新循环变量i受到判断条件“<=”和“！”控制，每 4 拍SUM才能得到一个i值，流水线执行的吞吐率为 4 拍一个循环迭代. 影响流水线吞吐率的第 2 个因素是循环调度与判断循环结束条件的

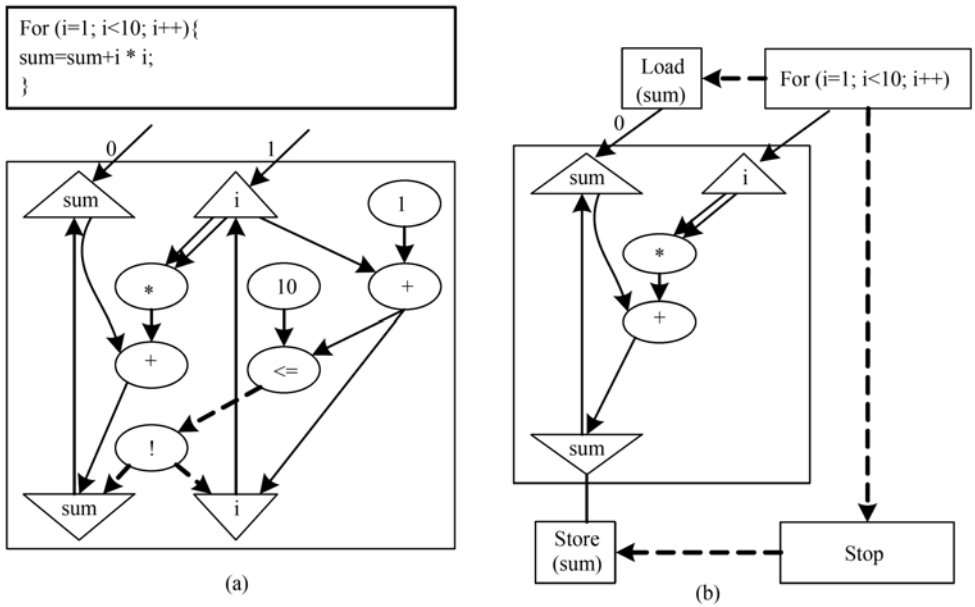

图 4

(a)累加例子程序和Spatial Computation[12]中的数据流图; (b)基于For 循环控制和猜测执行的数据流图

紧耦合同步关系. 如图 4 所示, i 值的更新必须受控于“<=”和“!”的测试结果, 目的是保证 i 值不会越界.

针对常用的多重 For 循环, 我们设计了集成循环变量计数器的分布式循环控制器, 结合循环步进状态机, 直接用硬件计数器实现循环控制变量, 保证每拍可以更新一次 i 值; 同时将集中的循环控制分布化, 每个取数部件和存数部件, 有独立的循环控制器, 可以直接提供循环变量值, 通过地址产生部件, 计算有效地址, 发出存储请求, 如图 4(b).

针对循环调度紧耦合同步, 我们提出了具有猜测执行机制的循环流水线执行方法, 将循环调度与判断循环结束解耦合. 循环调度方法是在接受循环结束条件为“真”之前, 以每拍一个循环迭代的速度更新循环控制变量, 受循环控制变量更新的驱动, LD 部件会不断地发出取数据命令(相当于数据预取), 返回的数据驱动计算单元执行. 当 LD 输出队列受阻, 暂停循环控制变量更新; 当接收到循环结束条件为“真”, 循环控制器停止输出新的循环控制变量值. 对于存结果到存储器, ST 部件受到严格控制. 其发出写存储器请求的条件必须满足“数据有效”且“地址有效”且“判断循环停止为假”. 循环结束判断部件通过 MPE 同步链向其他 MPE 发出循环每次迭代判断循环结束的结果. 每次从 MPE 中制定一个作为主控循环结束的判断, 它的输入与循环执行过程中条件判断相连, 相当于 Break 指令. 通过 Break 指令, 可以实现各种 While 循环; 对于规则的 For 循环, 直接用循环控制器就可以判断循环结束条件.

猜测执行也带来了副作用. 首先, LD 部件会持续预取数据, 仅当数据队列满, 才暂停. 其后果是在循环结束后仍然有数据存储在各个处理单元的数据队列中; 其次, 当循环结束判断部件接收到“循环停止为真”的信号后, 各个 MPE 的输出队列中仍然可能有有效的写请求没有完成. 所以循环判断部件在接收到“循环停止为真”的信号后, 进入到等待各个 MPE 完成有效写的状态, 仅当“循环停止为真”且所有 MPE 写请求被完成后, 发出循环执行结束信号, 同时清空所有 MPE 和 CPE 的队列.

3.2 支持数据重用的特殊数据传递指令

循环自动流水线实现每拍一个循环迭代的执行速度, 对存储器带宽的需求更大, 为了减少对存储器带宽的需求, 我们针对循环中不同的数据依赖关系, 设计了多种数据重用方法. 对于循环迭代内部的写后读和输入数据依赖关系, CPE 之间通过直接通道连接, 支持生产者直接传递数据到消费者; CPE 还提供了数据复制指令, 支持从输入复制数据到两个输出.

对于循环迭代间的写后读数据依赖关系, CPE 支持数据选择指令<C=select (A, B, *n*)>, 其语义为前 *n* 个数据 C 从 A 选择, 其后从 B 选择. 如图 1(b)中, 在对 A[i]的赋值与对 A[i–1]的引用之间存在迭代间写后读数据依赖关系, 数据相关距离为 1. 在第 1 个迭代, A[i–1]从存储器中取数据, 在其后的迭代中, A[i–1]可以直接引用A[i]在上一迭代产生的结果, 图中边⑴表示了这种引用和选择关系. 配置 CPE 为 C=select(A,B,1)就可以实现图中 J 的功能.

对于循环迭代间的输入数据依赖关系, 我们设计了非平衡广播指令<NB-Copy (Ain, Bout1, Bout2, BN1, BN2)>, 其语义为 Bout1 从 Ain 选择输出, 但是丢弃前 BN1 个数据; 同理 Bout2 从 Ain 选择数据, 但是丢弃前 BN2 个数据. <NB-Copy (Ain, Bout1, Bout2, 0, 0)>完成了前面提到的 1-2 数据复制功能.

在经典的编译技术中，研究人员 [13]提出了“寄存器提升”思想，利用循环间迭代输入数据重用，将被重用数据分配在寄存器中，通过每次循环迭代中旋转寄存器，达到寄存器中的数据在下一迭代中被重用，进而减少对存储器的访问．在文献 [12]中，提出了增加标志寄存器的方法，用以在动态执行过程中判断是否旋转寄存器．上述方法的主要缺点是需要在原程序中插入旋转寄存器代码段，同时要评估所需要的寄存器数量是否能够满足．

我们利用非平衡广播指令完成初始数据布局，同时利用 CPE 中 FIFO 队列的自动压栈操作完成执行过程中数据的旋转．如图 5(a)所示，源程序描述了取 3 个数组 A 的元素计算数组 R 的循环程序，其数据流图描述了计算过程．图 5(a)上部分的椭圆型方框表示使用数组 A 的顺序．对于 A[i]数据被引用的顺序为 A[1]A[2]A[3]…，A[i+1]数据被引用的顺序为 A[2]A[3]A[4]…，A[i+2]被引用的顺序为 A[3]A[4]…，从中我们发现从 A[3]之后数据都会被 3 个 LD 所引用，图中的斜线箭头标示了数据重用的规律．利用这一规律，首先将 A[1]送入 LD A[i]的第 1 个位置，然后将 A[2]送入 A[i]的第 2 个位置和 A[i+1]的第 1 个位置. A[3]之后的数据都同时送到 3 个 LD 输入队列．这样能够将原来每次迭代取 3 个操作数，减少到每次只取 1 个操作数．图 5(b)表示了采用非平衡广播指令实现初始数据布局的方法，其中 CPE0 使用通常的 1-2 数据复制指令，A[i]按顺序接收 A[1]A[2]A[3]…; CPE1 从输入端接收到 A[1]A[2]A[3]…，由于使用了非平衡广播指令，送往左端的第 1 个数据被丢掉，如图中的 A[1]虚线椭圆框，类似的右端的前两个数据被丢掉，构成了图5(a)所要求的初始数据布局，其后从 A[3]开始，数据被连续复制，不会被丢弃．在图 5(b)中还可以看到，数据在 CPE 的输出数据队列 FIFO 中，按照先入先出规则，到达计算 CPE，由于 CPE 严格按照数据驱动的顺序弹出操作数，操作数到达计算 CPE 的相互关系如图 5(b)的斜横线所示，并且每个迭代均匀下移，能够保持与图 5(a)的语义一致．

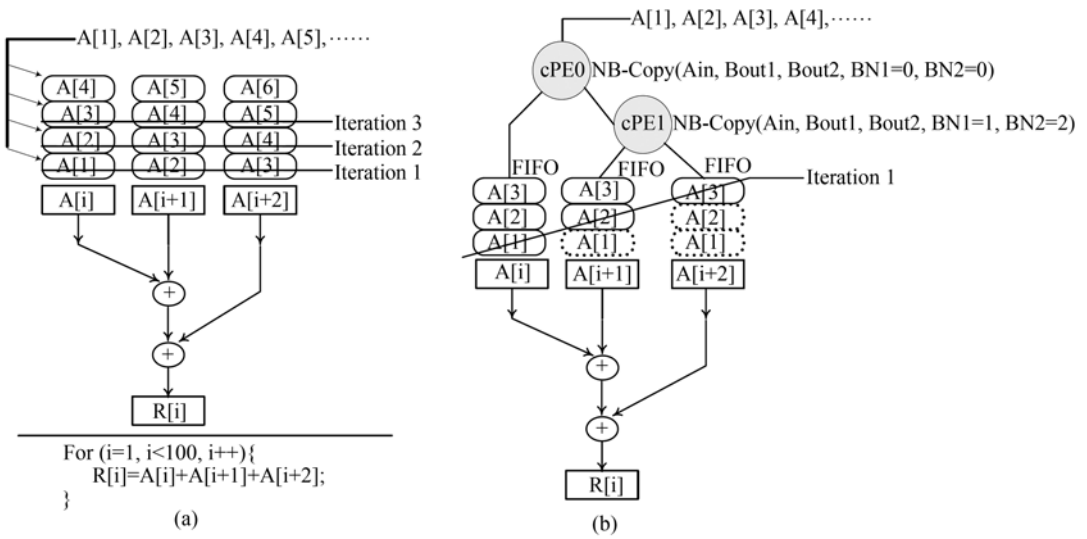

图 5

(a)具有输入数据相关的例子程序和执行过程中数据重用的顺序; (b)采用非平衡数据复制指令，完成同样数据重用的顺序

通过 CPE 支持数据直传、数据复制、数据选择和非平衡广播等指令, LEAP 体系结构提供了充分利用循环迭代内部和循环迭代间数据依赖关系减少存储访问次数的有效手段, 为提高循环流水线的实际吞吐率创造了条件.

4 试验与性能分析

我们用 Verilog 硬件描述语言实现了 LEAP 体系结构的所有模块, 并且在 FPGA 开发板上实现了不同规模的可重构阵列. 该实验环境包括 Altera 公司的大容量 FPGA 芯片 Stratix EP2S130F1020C6, 32MB SDRAM 和 PCI 桥. FPGA 芯片与 SDRAM 之间有 64 位的数据通道, 主控处理器(PC 机)通过 PCI 总线与 FPGA 芯片进行通信. 首先, 该实验平台能够运行实际应用程序, 提供时钟精确的实际运行结果. 其次, 可重构阵列的计算直接与外部存储器交换数据, 能够评估存储访问优化方法的实际效果. 同时, 通过在 FPGA 上的逻辑实现, 能够分析体系结构设计中各个模块的硬件资源使用情形, 综合工具采用了 Quartus II 6.0. 为比较 LEAP 存储优化指令的效果, 我们将测试程序在 SimpleScalar 模拟器上进行了分析. 该模拟器模拟了 4 发射、乱序指令流出的超标量 RISC 处理器, 其他配置包括 32 KB 一级数据 cache, 4 个整数部件、4 个浮点部件和 2 个外部存储器端口. 测试程序如表 1 所示, 其中 Frame 为一幅图像, 包含 15840 个宏块, 约 6 MB 数据.

表 1 测试程序

测试程序	规模	算法
Median	320×240, 480×360 图像	图像中值滤波
Sobel	320×240, 480×360 图像	Sobel 边缘检测
FFT	512 点, 1024 点	快速 Fourier 变换
MM	64×64, 128×128 矩阵	矩阵乘
Fdct	1 Frame	前向离散余弦变换
Quant	1 Frame	模拟量数字化过程中的量化编码

LEAP 阵列在 FPGA 上综合结果见表 2. 表中分别给出了 MPE, CPE 和 6 输入、6 输出路由单元的综合结果, 以及组合成 8×8 阵列的综合结果, CPE 中实现了 16 位整数运算部件. 后 4 列的数据分别为综合后各部件占用的逻辑单元、存储器、DSP 乘法器数目以及频率. 从表 2 的时钟频率数据可以看出细粒度 FPGA 在性能上的不确定性, 在一个 PE 的情形下, MPE, CPE 和路由器的时钟频率都能达到 100 MHz 以上, 但是当 64 个 PE 集成在一起时, 由于存储资源、连线资源冲突, 系统总体频率仅能达到 53.65 MHz, 从中可以看出可重构体系结构向粗粒度发展的必要性.

表 2 FPGA 综合结果

部件	逻辑单元	存储/bit	DSP	频率/MHz
MPE	778	5536	12	133.46
CPE	758	3104	2	106.85
路由单元	286	0	0	214.87
8×8 阵列	63255 (60%)	1809304 (27%)	202 (40%)	53.65

表 3 第 6 列的#CPE 表示参加计算的 CPE 数量, 因为测试程序的循环体规模不同, 实际参加计算的 CPE 数量也不同, 最多的用了 30 个 CPE, 最少的用了 10 个 CPE. 参加计算的 CPE 数量决定了循环流水线的最大吞吐率. 发出循环迭代的速率是影响循环流水线吞吐率的另一个重要因素, 由于循环流水线采用了猜测执行机制, 测试程序都取得了每拍发出一个循环迭代的性能, 因此在第 6 列中, 每拍最大吞吐率#OP 等于#CPE.

表 3　性能数据

测试程序	规模	LEAP						SimpleScalar		
		AP/%	#L	#T	#CPE /#OP	Rtime/K	#MAN/K	#IPC	Rtime/M	#MAN/M
Median	320×240	42.30	54	12.7	30/30	220.0	598	1.88	41.86	38.64
	480×360	41.20		12.4		478.8	1200	1.88	94.67	87.32
Sobel	320×240	41.47	54	6.6	16/16	217.0	598	1.85	9.94	7.55
	480×360	40.62		6.5		474.2	1200	1.84	22.52	17.05
FFT	512	40.26	34	4.0	10/10	6.7	25.6	1.36	3.69	0.21
	1024	43.49		4.4		12.8	51.2	1.38	7.35	1.90
MM	64×64	41.62	24	6.7	16/16	79.1	279	1.88	4.77	3.51
	128×128	82.60		13.2		580.3	4456	1.50	42.82	27.66
Fdct	1 frame	39.17	24	7.1	18/18	2839.0	27394	1.68	2369.38	1292.50
Quant		32.58	64	5.5	17/17	2131.6	2395	1.05	373.40	157.34

最大流水线吞吐率只是实际吞吐率的上限, 有多种因素影响着流水线的实际吞吐率#T. 其中之一是流水线的启动开销, 也就是从启动第 1 个循环迭代到第 1 个计算结果到达存储器的延迟时间. 在 LEAP 体系结构中, CPE 运算流水线级数、经过的路由器数量以及输入输出队列延迟构成了循环流水线的启动延迟. 从表 3 第 4 列#L, 可以看出启动延迟处于 24 至 64 拍之间, 相比与每个循环 1000 量级的总运行拍数, 启动延迟不是主要性能瓶颈.

影响流水线实际吞吐率#T 的最主要因素是数据在外部存储器和内部存储器之间的数据传输延迟. #AP 表示 CPE 计算活跃率, 也就是 CPE 计算时间占总运行时间的百分比. 虽然我们在测试程序中考虑了计算与存储访问之间的时间重叠, 但是存储访问延迟仍然降低了循环流水线的实际吞吐率. 在我们的测试中, 128×128 规模的矩阵乘程序取得了82.6%的计算活跃率, 主要是因为矩阵乘算法具有输入 $O(n^2)$、计算 $O(n^3)$的特点, 而其他程序都属于$O(n)$算法, 所以最小的计算活跃率仅为 32.58%, 平均为 45%, 因此存储访问的优化对于提高循环流水线效率非常重要.

表 3 第 8 和 11 列的#MAN 分别表示在 LEAP 和 SimpleScalar 下发出存储访问的次数, 从中比较出 LEAP 采用存储优化指令后减少存储访问次数的效果. 由于 SimpleScalar 的编译器没有进行寄存器提升优化, 仅有有限的寄存器优化, 它产生的存储访问次数远远多于 LEAP. 在执行 Median 滤波程序中, LEAP 存储访问次数减少得最多, 仅为 SimpleScalar 的 2%, 全部测试程序平均存储访问次数也仅为 SimpleScalar 的 3%. 存储优化指令结合多 CPE 流水线并行导致测试程序在 LEAP 上的执行节拍数远远小于 SimpleScalar 代表的超标量 RISC 处理器, 执行节拍数(Rtime)最大相差 2 个数量级以上. 另外, #IPC 表示平均每拍指令执行数.

为进一步验证LEAP体系结构对于复杂应用问题的适用性和有效性，我们在LEAP阵列上实现了一种基于Hausdorff距离的静态图像匹配系统，利用Hausdorff距离衡量图像间的匹配程度. 图像匹配是使用包含目标的模板在图像中搜寻是否存在模板中的目标，并对目标进行标识，包括滤波去噪、边缘检测、二值化、距离变换和平移匹配等多种算法，LEAP 阵列通过多次重构过程完成整个静态图像匹配. 在实验中采用了 48×48 像素的模板和 320×240 像素的图像进行图像匹配, LEAP阵列经过一系列算法的处理后能够精确地识别出模板在图像中的位置，其执行拍数与微机(主频 3.06 GHz 的 CPU, 1GB 的 DDR2 内存)相比能够达到 34 倍的加速比.

5 总结

LEAP 体系结构采用了数据驱动的设计思想用于分布式循环控制，提高循环流水线适应动态条件变化的能力；结合数据相关关系，提出了多种数据传递指令，充分利用数据重用性，减轻高流水线吞吐率对存储带宽的压力. 我们将LEAP体系结构在FPGA试验板上进行了实际映射，通过测试程序的实际运行，验证了 LEAP 体系结构的正确性. 在考虑外部存储器带宽和内部存储器容量受限的条件下，测试了循环自动流水线的计算效率，以及存储优化指令的效果. 通过在静态图像匹配中的应用，进一步验证了 LEAP 体系结构对于复杂应用问题的适用性和有效性.

参考文献

1 Cardoso J M P. Dynamic loop pipelining in data-driven architectures. In: Bagherzadeh N, Valero M, Ramírez A, eds. Proc 2nd Int Conf Comput Front. New York: ACM Press, 2005. 106—115

2 Baumgarte V, Ehlers G, May F, et al. PACT XPP-A self-reconfigurable data processing architecture. J Supercomput, 2003, 26(2): 167—184 [DOI]

3 Mei B, Vernalde S, Verkest D, et al. Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling. In: When N, Verkest Di, eds. Proc Conf Des, Autom Test Eur. New York: IEEE Press, 2003. 10296—10301

4 Barat F, Jayapala M, Beeck P O. Software pipelining for coarse-grained reconfigurable instruction set processors. In: Proc IEEE Int Conf VLSI Des. New York: IEEE Press, 2002. 338—344

5 Hauser J R, Wawrzynek J. Garp: A MIPS processor with a reconfigurable coprocessor. In: Proc IEEE Int Symp Field-Programmable Custom Computing Machines. New York: IEEE Press, 1997. 16—21

6 Rau B R. Iterative modulo scheduling: An algorithm for software pipelining loops. In: Proc ACM 27th Int Symp Microarchitecture. New York: IEEE Press, 1994. 63—74

7 Lee M H, Singh H, Lu G, et al. Design and implementation of the MorphoSys reconfigurable computing processor. J VLSI Signal Processing-Systems for Signal, Image and Video Technology, 2000, 24: 147—164

8 Dennis J B, Gao G R. An efficient pipelined dataflow processor architecture. In: Proc ACM Int Conf on Supercomputing. New York: IEEE Press, 1988. 363—373

9 Arvind, Nikhil R S. Executing a program on the MIT tagged-token dataflow architecture. IEEE Trans Comput, 1990, 39(3): 300—318 [DOI]

10 Iannucci R A. Toward a dataflow/von Neumann hybrid architecture. In: Siegel H, ed. Proc Int Symp Computer Architecture. New York: IEEE Press, 1988. 131—140

11 Kahn G. The semantics of a simple language for parallel programming. In: Sweden S, Rosenfeld J L, eds. Proc IFIP Congress. North-Holland, 1974. 471—475

12 Budiu M. Spatial computation. CMU CS Technical Report CMU-CS-03-217. 2003

13 Carr S, Kennedy K. Scalar replacement in the presence of conditional control flow. Softw Pract Exp, 1994, 24(1): 51—77 [DOI]

附录 A

用 Kahn 进程描述的虚拟指令执行语义.

```
Process F(input channel I; output channel L, R)//Data fork
Begin Channel-Data-Structure T;
   Repeat Begin
         T=wait (I);
         Send T on L;
         Send T on R;
         End;
End;
Process J(input channel L,R,P; output channel O)//Data join
Begin Channel-Data-Structure TL, TR, TP;
   Repeat Begin
         TL=wait (L);
         TR=wait (R);
         TP=wait (P);
         if TP then Send TL on O
         else send TR on O;
        End;
End;
```

```
Process LD(input channel: Address-in, EXE-token, Return-data;
              output channel: Address-out)
Begin Channel-Data-Structure T1, T2;
   Repeat Begin
         wait (EXE-token);
         T1:=wait (Address-in);
         Send T1 on Address-out;
         T2=wait (Return-data);
        End;
End;

Process ST(input channel: Address-in, EXE-token, Result-data;
              output channel: Address-out, Result-mm)
Begin Channel-Data-Structure T1, T2;
   Repeat Begin
         wait (EXE-token);
         T1:=wait (Address-in);
         T2:=wait (Result-data);
         Send T1 on Address-out;
         Send T2 on Result-mm;
        End;
End;
```

```
Process Compute (input channel: DataL, DataR;
                 output channel: Result; function f())
Begin Channel-Data-Structure TL, TR:
  Repeat Begin
            TL=wait (DataL)
            TR=wait (DataR)
            Send f(TL, TR) on Result
  End;
End;


Process MM (input channel: LD-Address, ST-Address,
                ST-Result; output channel: Return-data)
Begin logical B; Integer Memory[N];
B:=TRUE;
   Repeat begin
      if B then begin
        T=wait (LD-Address);
        send Memory[T] on Return-data
        B=~B;
      End
      Else begin
        T=wait (ST-Address);
        R=wait (ST-Result);
        Memory [T]=R;
        B=~B
      End
    End
End
```

```
Process G(input signal: Loop-start; input channel: Loop-end;
            output signal: Loop-over; output channel: EXE-token)
Begin Logical B; Channel-Data-Structure T1, T2;
   Repeat Begin
    B=wait (Loop-start);
    If B then Begin
     Repeat Begin
                Send TRUE on EXE-token;
                T=wait (Loop-end);
                if T. exe-token then Break & send
                TURE on Loop-over
                else send TRUE on EXE-token
              End
    End;
 End;
End;

Process E (input channel: Loop-check;
            output channel: Loop-end)
Begin Channel-Data-Structure T;
  Repeat Begin
            T=wait (Loop-check);
            if T.exe-token then send TRUE on Loop-end
            else send FALSE on Loop-end
  End;
End;
```

```
Primitive-op wait (FIFO A)
Begin
    Repeat begin
         If (~A. Empty) wait:=pop (A);
         End
End
```

```
Primitive-op send (FIFO A; Channel-Data-Structure T)
Begin
    Repeat begin
         If (~A. Full) push (A, T);
         End
End
```