多核及相关软件方法综述

王金德 20721156

2008-01-06

目 录

第 1 章 概述

在多核技术流行以前，处理器设计以摩尔定律[1]为动力快速发展。其主导方向主要集中在日益复杂的特性集，更高的时钟频率，不断增长的热温范围和更大的功耗等方面[2]。不过，摩尔定律本身不是亘古不变的真理。随着物理性质、功耗、设计复杂度等各方面的限制，在处理器设计已有的方向上实现突破已经非常困难。所幸的是，当把多个处理器（多核）集中在一个芯片上，增加相关的通讯同步等机制，利用它们的并行性和资源的冗余性，往往能实现比单核更高的性能和更大的灵活性。

多核技术成为当前处理器设计技术的主流，随之而来的是大量新的机会和挑战。随着多核技术的不断发展，其本身的研究方向也不断细化交融，每个方向都对多核技术的应用有着不同程度的影响。论文讨论了多核处理器设计的趋势，其面临的挑战，和相关的研究方向。论文着重研究了实现多核并行性的软件方法，从软件方法实现的目标不同把它们分为两类：细粒度并行软件流水和粗粒度并行软件流水。两者的界面有时并不十分明确，但是从根本上看，无论是实现的目标，采取的技术，利用的多核特性，对硬件的依赖程度等都有较明显的区别。

论文根据国内外多核的研究现状，着重于相关的软件方法，力求较为全面地综述。

第 2 章　多核技术趋势和挑战

2.1　多核技术与传统处理器技术

在多核成为主流的背景下，传统的处理器技术仍然有着一定的发展，但其面临的问题也是巨大的，主要包括如下方面[2]：

- 处理器时钟频率提高是处理器性能提高的一个重要方向。但是其指数级的增长已经因为功耗和设计复杂度，在4GHz左右处于平滑状态。
- 功率密度随着处理器特性增长也曾经呈现指数级增长。但是由于功率和密度在物理上都无法无限制地增长了，所以功率消耗和功率密度的增长目前都已经趋于平滑。
- 电压缩放技术曾被提出来，缓解功耗问题。该技术最终未取得应用，因为电压域值的改变需要更好的泄漏控制和全局静态能源消散，这本身带来的问题甚至超过了其带来的好处。
- 设计复杂度的增长已经接近了设计者可以承受的极限。摩尔定律的驱动力已无法给设计者带来足够的信心。
- 在一些特殊的应用领域，如低端的或嵌入式领域，虽然有着微结构、应用的特殊性所带来的好处，使用它可以简化甚至忽略处理器设计的某些方面，但是其特有方面的发展也有着很大的限制。

多核技术正是在这样的条件下，给处理器的发展注入了新了活力。从多核的自身的特点来看，它与前面提到的处理器技术的各个方向相比，主要有如下三大优势：

- 并行性；
- 自身资源固有的冗余，能够产生极其产生灵活的架构；
- 与分布式体系架构相比，多核技术由于把所有的核集中在一块芯片上，所以能允许低延迟的交互；

另外，多核技术发展的市场驱动力也是强大的。高性能计算及到普通计算对于处理器性能高速增长的需求是巨大的，同时，由于其他处理器技术所面临的难以克服的问题，多核技术成为了处理器发展的希望和潮流。

2.2　多核技术介绍

多核技术的发展和应用日新月异，但从根本上来说，多核技术主要有如下四种形式[2]：

- 异类的核（dedicated heterogeneous cores）综合在一起，提供多样化的应用。这种形式中，不同的核针对不同的方向，其整体构成高性能的复杂应用。
- 大量同类的互助核（remedial homogeneous cores）分担所有的计算。这种形式一般用于低端或嵌入式应用，因为这些应用往往每个核的能力是极其有限的。
- 少量复杂的同类核（complex homogeneous cores）结合在一起，每个核独立地完成或者提供几个辅助应用以计算资源。
- 当然，前面三种方式可以不同程度地结合在一起，构成复杂的应用，而这正是第四种形式。

不同的应用采用不同的多核形式，而一般在其中做出选择的标准是：

- 提供最大的吞吐量；
- 满足DFM/DFY要求；
- 满足热量和能耗封装；
- 满足成本要求；
- 可验证性；

这些并不意味着多核技术是一个相对孤立的技术，它与其他相关技术也有着密切的联系，比如多核上的应用软件设计、HW/SW联合设计、IP核的重用等。多核技术也带来了新的设计复杂性，包括核间通讯和缓存访问策略等。

2.3　多核技术的挑战

多核技术给处理器设计带来巨大的发展的同时，也给设计者带来了重多新的挑战[2]：

- 非核心或粘合逻辑的设计；
- 保持Cache的一致性，管理Cache在命中和失配时的访问；
- 重用技术；
- 后期硅制品(post silicon)功能和性能验证的复杂度增加，测试难度增加；

- 随着多核中核的指数级增长, 核软件在多核领域起了更重要的作用；
- 微架构(micro-architects) 增加了错误率，这就要求更强的自我检查，错误检测等方面的设计；
- 3D硅方法带来新的优势和挑战；
- 核间通讯机制的挑战，多种核间交互策略和协议的选择；

并且，多核技术在某些方面，也存在和传统的处理器技术同样的问题，比如多核要求测试、诊断和自我修复必须考虑在整体设计之内，增大了设计的复杂度；多核技术增加的处理特性重新带来功耗和功率密度指数增长的危险等。

2.3.1　核间通讯机制

核间通讯机制多核体系架构中起着越来越重要的作用，这也正是实现多核的一个重要挑战。

核间通讯主要可通过核间互连和协议实现。不同的核间通讯微架构导致不同操作的延迟也有所不同，这样需要为重要的功能选择一个更短延迟的架构。而这种选择的衡量标准本身也是一个挑战。

多核相对于单核或多处理器的分布式架构来说，其核间互连通讯机制的设计是一个重要内容。这种通讯机制影响着多核的各个方面，而本节也正是以这个为切入点，讨论多核的核间通讯机制。

芯片内多核互连结构与其他如芯片间、多芯片模型、板级结点等互连结构不同的原因在于：

- 对于芯片内多核互连，功率、区域、延迟和带宽都是第一要素。
- 在设计中, 核、缓存和互连结构更高层次地结合在一起，互相影响，互相制约。

芯片内互连体系结构类型主要包括交叉开关、点对点连接、总线架构、以及这些技术不同程度的结合。而事实上，这几种结构常常针对的是互连结构的不同方面，这也是它们常常能有机结合在一起，共同发挥作用的原因。目前已经实现的核间互连结构有Piranha的芯片内开关（Intra-Chip Switch）[3]，Hydra的交叉开关（Crossbar）。这两种的二级Cache是完全共享的。另外IBM Power4实现了一种和交互开关类似的结构：CIU（Core-Interface Unit）[4]。

针对互连体系结构的三种基本类型，论文[4]给出了比较详细的描述，并提出

了三种相应的互连机制。

- 共享总线结构（Shared Bus Fabric，SBF）是一种CMP系统内处理器、缓存、IO和内存之间传输数据的高速连接。主要由请求和数据队列、地址总线仲裁、地址总线、监听总线、响应总线和双向数据总线构成。它支持基于监听的（Snoop-Based）共享内存多核结构。该论文实现了与MESI类似的，二级缓存回写的监听写无效协议。所有的总线都是单向的，所有的核都拥有私有的一级和二级缓存，而SBF连接二级缓存来保证内存请求和一致性。如图2.1。

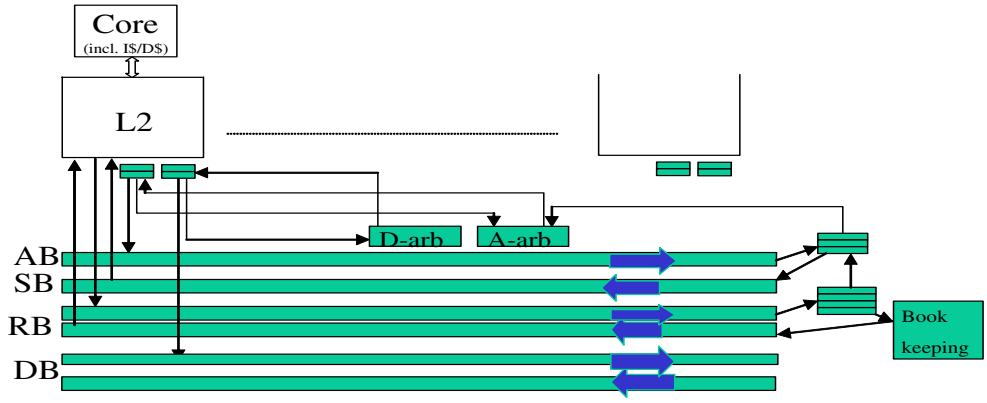


图 2.1　一种共享总线结构[4]

- 点对点连接（Point-to-Point Link，P2P Link）是用来连接不同的SBFs，支持它们之间互相的通讯。它能够双向传输所有种类的事务（请求、响应、数据），并且P2P连接的终点是多重队列。如图2.2。

- 交叉开关互连系统（Crossbar Interconnection System）由交叉开关连接和交叉开关接口逻辑组成。它常用来完成多个核共享二级缓存时，核间的通讯。一个交叉开关通常由一条地址组和两条数据线组成。如图2.3。

在设计核间通讯机制时，除了考虑性能之外，常常还需考虑多种开销，一般包括配线区域和逻辑区域开销、功率开销和延迟开销等。

总之，在多核中，核内互连体系结构和核体系结构、缓存体系结构紧密地结合在一起，必须从全局的观点来看。所以多核设计者要想取得好的设计，必须采用三者的联合设计。

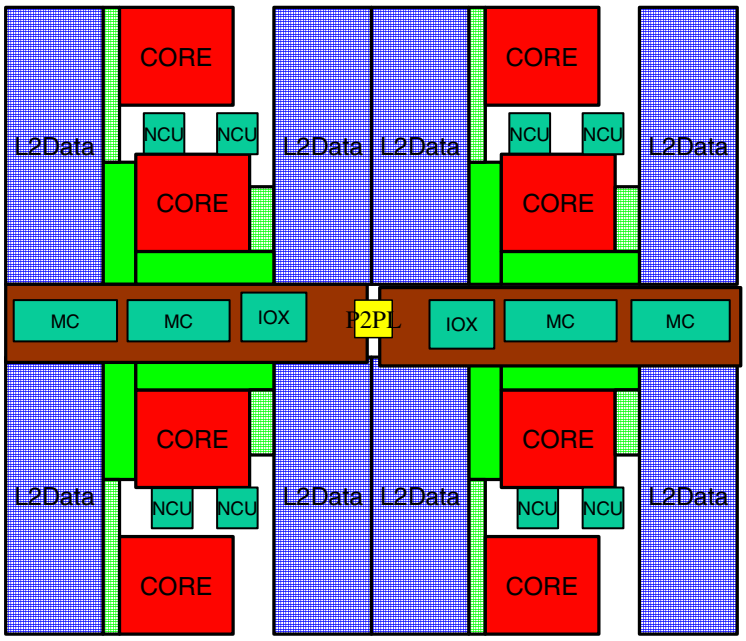


图 2.2　点对点连接[4]

2.3.2　多核自动化设计

自动化设计是解决设计复杂度问题的一个有效途径。完全实现自动化设计往往是一件很困难甚至于不可能的事情，而部分实现自动化设计也能给设计者带来巨大的好处。

在多核领域，自动化设计同样面临巨大的挑战。而从实现来看，主要可以从Post-RTL和Pre-RTL两个阶段入手[2]。

Post-RTL阶段的实现主要面临两大挑战：

- “时间闭包（Time Closure）”向“设计闭包（Design Closure）”的转化要求必须对处理器的功率、性能、成本和可靠性同时进行优化设计，而不能互相脱离。
- 设计最优化（Design Optimization）和可制造性（Manufacturability）必须综合对待。

Post-RTL阶段的实现方式主要体现在重用上，包括设计重用、验证重用、布局重用和测试重用。

Pre-RTL阶段主要解决在不同的设计中选择权衡的问题。这主要从两方面考虑：

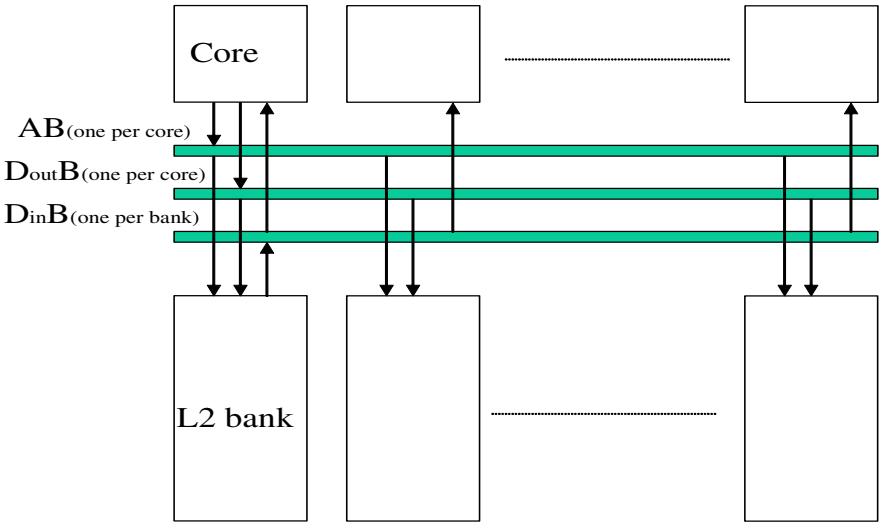


图 2.3　典型的交叉开关[4]

- 和Post-RTL一样，从重用性考虑解决。
- 使用参数模型、高层规范模型或系统建模语言来解决问题。具体需考虑的方因素有功率模型、物理模型、热区、早期综合分析系统、优化（主要是多变量优化）、交付实现、可靠性、三维封装等。如图2.4。

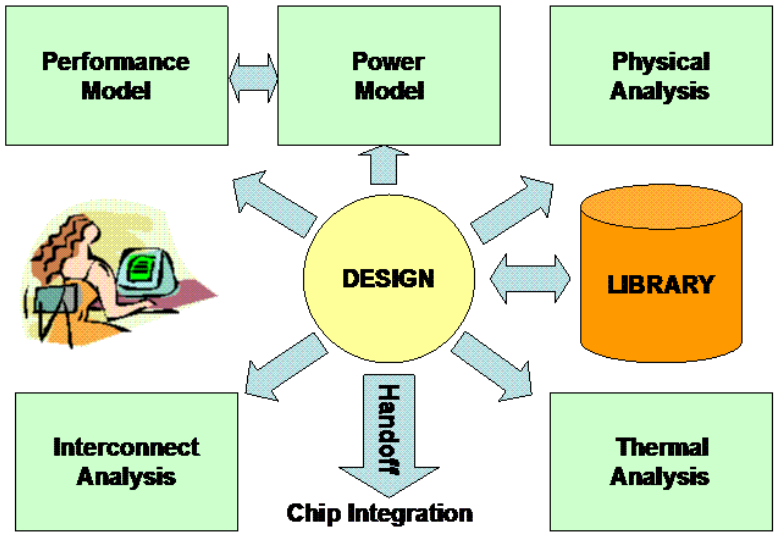


图 2.4　Pre-RTL综合分析[2]

第 3 章　细粒度并行软件方法

3.1　简介

多核技术发展离不开相应软件方法的发展。软件方法的进步，使得多核技术的优势得到极大的发挥，多核的并行性得到正确的利用，从而使得系统的性能有了很大的提高。在实际上，有各种各样的软件方法存在，这些方法从编译技术和运行时系统（如操作系统）等方向给予多核强有力的支持。

不同的软件技术利用不同的多核特性，以获得最大的性能。从软件方法实现目标来分，主要分为两种：细粒度并行软件方法和粗粒度并行软件方法。两者的界面有时并不十分明确，但是从根本上看，无论是实现的目标，采取的技术，利用的多核特性，对硬件的依赖程度等都有较明显的区别。细粒度并行每执行一条指令都要切换线程，多个线程交叉进行。这种交叉通常以一种循环的方式进行，并忽略被阻塞的线程。致力于这种并行的软件方法一般常常依赖于较多的硬件技术的支持。而粗粒度并行是除细粒度并行之外的线程级并行。致力于这种并行的软件方法往往无需相应硬件技术、体系架构等的支持，具有良好的可移植性。

在多核出现以前，指令级并行的软件方法已经广泛存在，并且在已存在的几乎所有的编译器中广泛应用。这些技术在多核环境下也具有一定的适应能力，但一般在多核上真正地应用，仍需很大的努力。

本章针对细粒度并行软件方法进行探讨。传统的软件流水技术基本上都属于这一类，同时，一种新的软件流水－－解耦软件流水[5] 虽然也支持一定的粗粒度并行，但总的来说，还是基于细粒度并行实现的。

3.2　传统软件流水

传统软件流水技术主要从流水线调度和循环展开两方面利用指令级并行，而两者往往能较好结合在一起。传统软件流水在单核上的成功应用，可以给多核上的软件方法带来很多经验和启发，甚至是一定程度上的重用。

对于传统软件流水，本节并不想过多讲解，因为这种技术已经相当成熟。本节关注于它在多核上的应用，以及对多核软件流水研究所起的作用，同时对一些

基础知识作一些简单的描述。

软件流水在关注实现并行本身之外，最关注是解决依赖问题。这些依赖和处理器设计中的依赖有很大的相适性。下面几种依赖往往是软件流水中常常需处理的[6]：

真依赖 又叫数据依赖。指令j真依赖于指令i有两种情况：指令i产生结果将被指令j使用；指令j真依赖于指令k，而指令k真依赖于指令i。

输出依赖 指令i，指令j有输出依赖：指令i，指令j写同一个寄存器或内存区域。

反依赖 指令i，指令j反依赖：指令j要写指令i正在读的寄存器或内存区域。

在软件流水技术中，大部分都与处理循环相关，而循环也在研究中被不断细化、分类。而不同的流水技术也常常针对不同种类的循环来进行循环展开。

循环常常可以分为嵌套循环和非嵌套循环，嵌套循环又可分为[7]：

完美循环 指在外层循环中，除了一个或多个内层循环，没有其他代码。

非完美循环 除了完美循环之外的所有嵌套循环。

在用于多核的传统软件流水里面，DOACROSS技术是其中较为著名的一个。DOACROSS技术通常指一类技术，在解释该类技术之前，需要了解一些相关的概念[7]：

交叉迭代依赖（Cross-Iteration Dependence） 指不同迭代之间的依赖约束。

DOALL循环 指所有的迭代之间都没有交叉迭代依赖约束的循环。

DOSERIAL循环 指存在交叉迭代依赖约束的循环。

DOACROSS技术正是一类通过一些并行运行不同迭代的处理器之间的同步机制，以遵守交叉迭代依赖约束的技术。

3.3　解耦软件流水

解耦软件流水（Decoupled Software Pipelining，DSWP）[5]是一种高效的、完全自动的非投机线程抽取技术。它开发了隐藏于大多数程序中的长周期、并发执行的线程。DSWP通过产生非投机的和真正解耦的线程，来提高执行效率，提供重大的延迟容忍，并通过简化核间通讯和各核的资源请求来降低设计复杂度。

虽然DSWP对线程进行了解耦，但是它是在指令级实现的，它所操作的都是已经编译过的指令。换句话说，它所针对的是细粒度并行，即指令级并行。DSWP同时支持粗粒度的人工线程化（Coarser-grained Manual Threading），投

机线程化（Speculative Threading）和预线程化（Prefetch Threading）技术。DSWP在指令级实现，但同时仍允许编译器实现指令级并行，能较容易地添加到当前存在的编译器的后端。但从总体上来说，它仍是属于细粒度并行软件方法。

DSWP中对线程的解耦，在某种程度上类似于软件工程中，软件架构的层次结构的解耦：传统的软件流水倾向于把每个功能块都分成很多条并行执行的流，所以各流之间有很多依赖（如DOACROSS技术）；而DSWP相当于把程序按“功能”进行划分成不同的层，每个层执行于不同的核，这些核间只有数值的依赖，这可以通过核间队列解决。如遍历一个链表并操作结点上的数值，指针增加（一个功能）和数据自增（另一个功能）被DSWP分离，分别在两个核上运行，核间仅有数据依赖，这样实现了线程的解耦。

3.3.1　DSWP优势和限制

从纯编译角度，指令级并行编译技术很成功，线程级并行编译却相对举步维艰。不过在科学和数值计算领域，专用的线程级并行编译技术也有一定应用。这些技术一般在处理简单的可数循环、可分析的结构、可预测的数组访问等方面取得了良好的性能，因为在这些特定的案例中，DOALL循环很普遍，或者很容易通过循环遍历转换得到。但是，在大部分普通程序里面，控制流、递归的数据结构、指针广泛使用，使得这些技术不能适应。另一方面，计算机体系架构师们通过一些投机和多重路径技术，在硬件环境下实现自动线程抽取。但是这种技术在发生投机失配或者需要对微体系架构产生影响时，通常需要显著的硬件支持；同时投机功能越强，投机失配率和失配处罚也会越大。这些说明了，本节的方法－－自动地非投机的线程抽取是不可替代的。

DSWP要求核间的数据流动必须是无环的[5]。这种无环的流程使得DSWP通过核间队列缓存核间交互的数据值，从来实现线程间解耦成为可能。这样不同的循环常常被赋予不同的核，因为没有循环依赖的存在。总得来说，DSWP通过解耦执行，更好地利用了并行资源，并且允许高级的延迟容忍。DSWP上述要求导致无法适应有些循环，不过论文指出，在采取了ILP优化之后，这类情况是很少的。另外，DSWP除了上述限制之外，它没有DOACROSS技术需要的其他限制：循环次数是可数的，数组单独操作的，有规则的存储访问模式，或者必须简单的控制流（甚至不允许有）。

DSWP技术与DOACROSS技术相比，两者最主要的区别是在处理循环中的关键路径的问题上。DOACROSS技术的关键路径扩展到多个核上，这样，其代价是平均核间通讯延迟X迭代次数；而DSWP技术关键路径依赖只存在于一个核中，所以基本没有显著的通讯延迟，当然它的代码是需要一个核间队列来缓存核间交互的数据值。如图3.1。

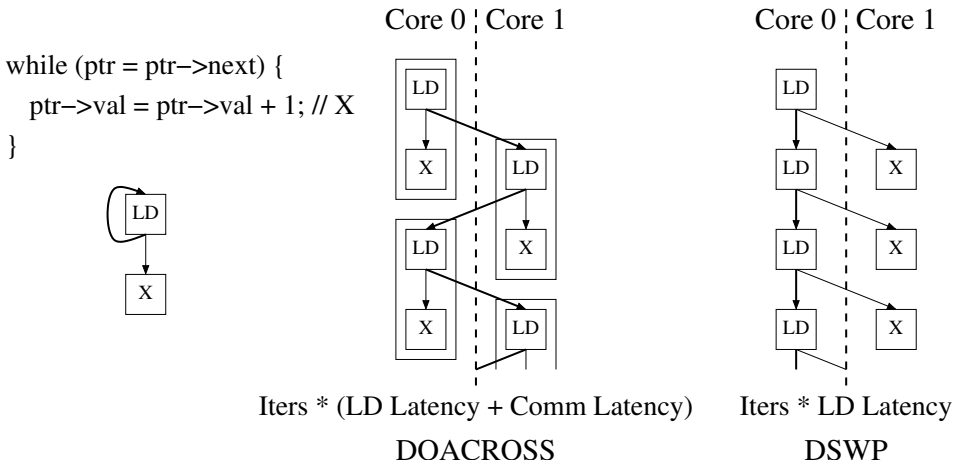


图 3.1　DSWP与DOACROSS在一个简单链表遍历上的区别[5]

3.3.2　DSWP技术分析

DSWP假定一个简单的消息传递机制: 每条消息只传递一个字，这可以由硬件高效的实现[8]。具体的通讯可由produce和consume两条指令实现，并且它们是成对按序出现的。DSWP中队列延迟不很重要，而同步开销则显得相当重要，因为它可能会降低关键路径循环的速度。为解决同步开销的问题，DSWP 要求produce和consume操作只在往满队列中插入或从空队列中取数据时才会阻塞; 这样的队列实现可在[9,10]中找到详细描述。

基本的DSWP算法分为四步：

1. 建立依赖关系图；
2. 线程划分;
3. 分裂代码；
4. 插入流（flows），即produce/consume对。

建立依赖图的算法详细见论文[11]。依赖图中包含所有的数据，控制和存储依赖，包括内部迭代和外层循环。对于寄存器数据的依赖，只要考虑真依赖，而不需考虑输出依赖和反依赖。

线程划分步骤包括：

1. 找到紧密连接成分（Strongly Connected Components，SCCs），创建直接无环图（Directed Acyclic Graph，DAGSCC）。
2. 如果只有一个SSC，停止转换。
3. 合并每个SSCs成为一个结点。
4. 真正处理线程划分问题（Thread-Partitioning Problem，TPP），该问题是选择有效划分，使用结果代码执行时间最短。谓有效划分，指DAGSCC是P1，P2，...，Pn的序列，而这些P又是DAGSCC中结点集合。同时需符合以下三个条件：
   - 划分数n大于1，且小于等于目标处理器所能同步运行的最大线程数t；
   - DAGSCC中的每个结点属于且只发球一个划分；
   - 对于DAGSCC中的每条边（arc，u-¿v），若u ∈Pi and v ∈Pj，则i¡=j。

   TPP是一个NP完全问题，可归约到二分包裹问题[12]。所以一般采用一种启发式算法（估计执行循环次数）来最大化线程间负载均衡。
5. TPP可能返回单一的划分，表示无任何划分满足要求，这时也停止。

根据线程划分的结果，进行分裂代码：

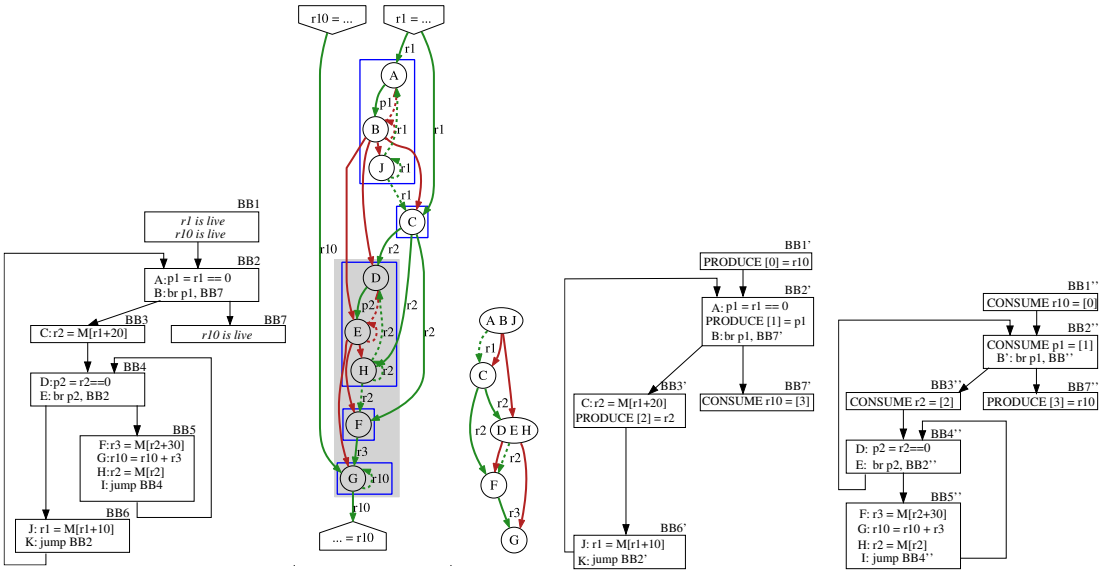
1. 计算相关基本块（BBs）集合;
2. 创建Pi的BBs；
3. 把划分给Pi的相关指令放入到对应的BB中，BB中的代码保证原来的顺序；
4. 修正分枝目标，每个Pi中的跳转指令可能会有重复。

接下来，插入流解决各种依赖关系。流基于的依赖类型有三种：数据依赖、控制依赖、内存/同步依赖。

流根据其相对于循环的位置不同，又可分为：循环流（Loop Flow）、初始流（Initial Flow）、内存/同步依赖（Final Flow）。

在插入流后，可进一步进行冗余流消除。

图3.2提供了一个具体的实例。程序(a)遍历一个链表，而该链表的每一项又是一个整数的链表，程序计算了所以整数的和。r1存储外层链表的指针，r2存储



(a) Original code　(b) Dep. graph　(c) DAG_{SCC}　(d) Producer thread　(e) Consumer thread

图 3.2　一个DSWP具体实例[5]

内层链表的指针，r10存储最终的核。两层链表都是以指针为0表示链表结束。

(d)核负责r1，r2指针的自增，(e)核负责指针指向的数据的数值的计算。两者之间的通讯使用了四条队列：

- 队列一：(d)把保存结果的r10放在队列中，即r10由(d)初始化，只在刚进入程序的时候使用;
- 队列二：(d)向(e)传递计算出来的r1值;
- 队列三：(d)向(e)传递计算出来的r2值;
- 队列四：(e)向(d)传递最终结果的r10，在程序结束时使用。

但两个核会去执行一些相同的指令，如对r1，r2是否为0的判断等，这应该会影响部分性能。

DSWP算法为保证其正确性，对传统控制依赖的内容作了一定的扩展，主要加了以下两种控制依赖：

循环迭代控制依赖（Loop-Iteration Control Dependences）：

DSWP中，队列被每个迭代重用，所以编译器需要保证来自不同迭代的值被正确的传递。这就需要产生一种循环迭代控制依赖。这种依赖本身是存在的，只是按照上文描述的DSWP算法执行时，会被忽略，论文通过概念性

地展开循环中第一次迭代，重新为DSWP生成这种依赖。

有条件的控制依赖（conditional control dependences）：

对于那些依赖关系的发生无法在编译时确定的，则需要传递依赖关系发生的条件本身，在这种情况下就需要加入有条件的控制依赖，保证程序正确。另外，在有外部数值时的循环退出中，也需要加入这种依赖。这种依赖的加入，可能会造成生成的SSCs个数减少，不过论文提到在实验中未碰到这种情况。

3.3.3　DSWP小结

DSWP的思想最早由[10]提出，但它是手动修改递归数据结构循环的代码实现。而本节描述的DSWP实现了更一般化的并行技术，提供了第一个编译算法，并完全大量自动化的测试实现。另外，DSWP中还可以引入一些投机行为，以提高其性能[13]。

不妨把DSWP与传统SWP进行更为细致的比较：

- SWP通过重新组级别循环指令实现指令级流水；DSWP划分和调度循环代码实现线程级流水。
- SWP在ILP技术上效率很高，但在可变的延迟指令存在时表现很差; DSWP能实现更好的延迟容忍，这归功于粗粒度并行和线程流水的解耦执行。
- SWP和DSWP能够同时被应用到编译技术中。

DSWP的灵感还来自于DAE（Decoupled Access-Execute Architectures），不过DAE不能有任何单个的线程先运行，也不能实现任何粗粒度并行; 而DSWP通过在线程流水创建阶段避免线程间的循环依赖解决了这个问题。

对于DSWP，还可以改进的方向有：更精确的存储分析；对打破依赖循环的额外优化；更精致的划分启发示算法; 旨在减少流（flows）数目的优化。因为DSWP与许多编译技术都是正交的，所以可以结合优化达到最佳性能。

第 4 章　粗粒度并行软件方法

4.1　简介

粗粒度并行软件方法一般指针对线程级（TLP）及以上并行的软件方法。它往往无需相应硬件技术、体系架构等的支持，具有良好的可移植性。粗粒度并行软件方法发展较细粒度并行软件方法晚得多，它是随着多核技术的发展而发展，其应用也不成熟。不过粗粒度并行软件方法明显更适合于多核技术。它能在高层次利用多核的并行性，这样，它就可以忽略一些底层的依赖、同步，提高性能；并且，粗粒度的并行也意味着更大的灵活性、可测性和伸缩性。

目前粗粒度并行软件方法主要包括两类：基于流语言的软件流水和线程级投机软件流水。本章正是分别探讨这两类软件方法，并分别从两类方法中的典型实现入手，以对它们有个清晰具体的认识。

4.2　基于流语言的软件流水

多核已经成为了处理器设计的工业标准，而传统语言如C、C++、FORTRAN等并不适应多核环境，因为它们都是单指令流和单内存的。这造成了多核资源和优势无法被很好地应用。而依靠编译器来解决这个问题又是极其困难的，因为目前的编译器并不能很好地实现多核上粗粒度的并行。而有一类语言－－流语言（Stream Language），因为其并行的特性，却能够较好地应用在多核体系中。

实际上，流语言最早被设计出来，并不是为了支持多核技术。它在很多应用程序领域，如网络，图像, 声音和多媒体等，得到了广泛的应用。但是流语言高层次粗粒度的并行性设计与多核提供的并发性能够天然地结合在一起，它们的结合是不可避免的。这也揭示出流语言应用于多核领域的一个核心的问题：怎么把流语言中丰富的并行性映射到多核体系结构中去？因为在结合这种并行性的同时, 常常同时带来通讯和同步开销，并且现实的应用程序中的每一条流程常常是不等价的，存在着一条关键流程。这诸多问题若不能很好的解决，两者的结合甚至得不偿失。

4.2.1　流语言

流语言最大的特性在于它的并行性。流语言的并行性主要可分为三类[14]：

任务并行（Task Parallelism）：

存在于原始流图（Stream Graph）的不同分枝上的执行单元（Actors）之间。这些执行单元中，任何一个执行单元的输出不会作为另一个执行单元的输入。在流语言中，任务并行指算法中的逻辑并行。这样，可以很容易地把任务并行中的各个任务映射到多核发中的各个处理器中, 在任务的端点（开始或结束处）分裂或合并数据流。任务并行的冲突在于分裂和合并数据流时的通讯和同步。另外, 任务并行的粒度取决于具体的应用（以及开发人员），所以仅仅采用任务并行是不够的。

数据并行（Data Parallelism）：

存在于一个执行单元中没有依赖关系的不同执行流程之间。这样, 一个无状态的执行单元能够实现无限制的数据并行，这些执行单元的不同实例可以跑在各种计算单元之上。数据并行很适合于向量机。但在粗粒度的多核发体系架构中，它会引进额外的通讯开销。以前的数据并行的流架构往往关注于设计一种特殊的针对于这种通讯的存储层次结构。数据并行的缺点在于容易导致缓存和延迟的增加, 并且不适用于那些有状态的执行单元。

流水线并行（Pipeline Parallelism）：

应用于在流图中直接相连的生产者和消费者链中。可以通过把生产者和消费者集群映射到不同的核中，再在不同的执行单元间使用芯片内互连实现直接通讯, 来实现流水线并行。和数据并行相比, 流水线并行提供更少的延迟和缓存，以及更好的局部性，也没有引入额外的通信, 同时提供了并发执行有状态执行单元对的能力。流水线并行的缺点在于引入了额外的同步, 因为生产者和消费者在执行中必须紧密地藕合在一起。另外, 还必须实现有效的负载均衡机制, 因为流图的吞吐量取决于所有处理器中吞吐量最小的处理器。

目前已经存在大量的流语言，如StreamIt、Brook、SPUR、Cg、Baker和Spidle等。各种流语言都有各自的特点，对于多核体系结构来说，StreamIt语言是一种比较

经典而合适的。本节讨论的一种基于流语言的软件流水也正是基于SteamIt语言。并且该技术依赖于StreamIt语言的两个特性：filters之间的生产者-消费者关系；整个流图被一个外层循环所包裹。

不妨稍微介绍一下StreamIt语言。

StreamIt语言[15]是致力于高性能流应用的体系架构无关的编程语言。StreamIt中的Filters对应上文提到的执行单元（Actors）。Filter中的work函数执行Filter中的每一步。在work中，filter可执行push到输出通道、pop输入通道、peek输入通道。StreamIt提供三种层次化的原语, 用于组织filters成为流图。这三种原语分别为：Pipeline、Splitjoin、Feedbackloop（该原语实际中很少使用）。

4.2.2　基于StreamIt的软件流水

简单地说，论文[14]实现了一个流语言（StreamIt）的编译系统, 把任务并行，数据并行和流水线并行结合在一起，实现了粗粒度并行，在软件编译方面利用了多核的优势，很大程度地提高了性能。

该编译系统主要实现了两个新的编译技术：

- 利用数据并行，通过增加流图的粒度来消除通讯开销。增加粒度通过改可能多地熔合执行单元实现。该技术还利用了任务并行，如两个平衡的任务并行执行单元只需并行运行在一半的核中。粗略估计，粗粒度数据并行取得相对单核的9.9倍的加速比，相对任务并行基准的4.4倍的加速比。

- 利用流水线并行。通过软件流水线技术来并行执行不同迭代中的执行单元，来避免同步机制的缺陷。传统上，软件流水线技术用于指令级并行。论文利用流编程的强大特性，把软件流水线技术用于粗粒度的并行。这有效地移除了流图定态迭代中的执行单元间的所有依赖关系，极大地增加了调度的自由度。和基于硬件的流水一下，软件流水允许有状态的执行单元并行执行。不过，软件流水没有同步开销，因为各处理器读写缓存，而不是直接地通讯。粗略估计，粗粒度的软件流水取得了相对于单核7.7倍的加速比，相对于任务并行基线的3.4倍的加速比。

数据并行在程序中广泛分布。编译系统通过简单地程序分析检测无状态filters。通过filter分裂，使得初始filter的定态work最终被分裂到splitjoin的组成

部分中。在多核体系架构中，从filter分裂的结果中分发数据是昂贵的。所以编译系统只分裂那些计算和通信比率大于某个给定域值的filters。为进一步减轻通信费用，编译系统引进两个新技术：粗化粒度和补充任务并行。

编译系统在把pipeline熔合为一个filter，再把这个filter分裂成数据并行的splitjoin时，会出现有些pipeline不能被完全的熔合的可能。因为有些熔合会引进内部状态，但是有状态的filter无法实现数据并行。所以论文提供的粗化粒度的算法是在保证熔合的结果filter是无状态的前提下，尽可能地实现熔合。

实际上，即使每个filter都是可数据并行的，也不能把每个filter都分裂到所有核上执行，因为这样会消除所有的任务级并行，效率反而不高。一种替代的方案是：保留原始程序中的任务并行，只引进充分的数据并行到那些空闲的处理器中。这样做的好处是降低了filter 分裂带来的同步开销，同时避免filter分裂过程本身的计算开销。编译系统设计了一种称之为智能分裂（Judicious Fission）的启发式算法实现。该算法自底向上地遍历流图，估计总工作开销，以权衡任务并级和filter 分裂之间的平衡。通过粗化粒度和智能分裂，实现了相对于原始分裂策略6.8倍的加速比。

每一时刻，流水线并行的执行单元各自执行各自的迭代，它们之间的数据交互通过队列保存。这也表明，两个执行单元执行的距离不能太远，否则它们交互的数据保存所需的缓冲会过大。这样，实际流水线并行在实现解耦每个执行单元的调度机制之外，还要实现限制缓冲区大小的机制。这可以通过硬件或软件来实现：

- 粗粒度硬件流水。主要面临两方面性能权衡: 映射连续的filters到一个给定的处理器；以动态、数据驱动的方式执行filters。前者严重限制了划分的选择条件，从而导致了糟糕的负载平衡; 后者导致通信模式不再静态，以及需要更复杂的流控制机制，并且动态分发本身也需要开销。

- 粗粒度软件流水。通过循环前序和定态循环两种调度实现。粗粒度软件流水好处是可以更自由的线程划分，有利于线程间负载平衡; 避免需求驱动模型的开销。

两种流水相比，粗粒度软件流水更灵活。而在程序中已经包含能够被很好的负载平衡的长流水线时，硬件流水往往效率更高。

软件流水算法映射filters到核的方式与传统算法中映射指令到ALUs中相似。该算法利用了StreamIt语言的一个重要特性：整个流图包裹在一个隐性的外部循环中。这要求考虑缓存管理和调度中的通讯，同时映射filters到核的过程与传统软件流水的过程也有所不同。在该方法中，每个filter读写缓冲，而不是直接通讯，这需要一个独立的通讯阶段来协调各缓冲中的数据。映射filters到核的目标是在通讯阶段最小化同步的过程中优化各核间的负载平衡。可通过两个阶段实现：先优化负载平衡; 再优化布局设计。前者是一个NP完全问题，采用贪心划分的启发式算法；后者用一个选择性的熔合包裹划分算法实现。

软件流水线与数据并行相比较而言：软件流水线无法降低瓶颈filter；数据并行更易于粗化流图的粒度，从而减少更多的同步。另一种考虑是软件流水线更依赖于静态工作估计策略的精确性，所以在测试上难以量化其作用。

一个基于StreamIt语言的软件流水实例如图4.1。它图示了任务、数据、流水线这三种并行。

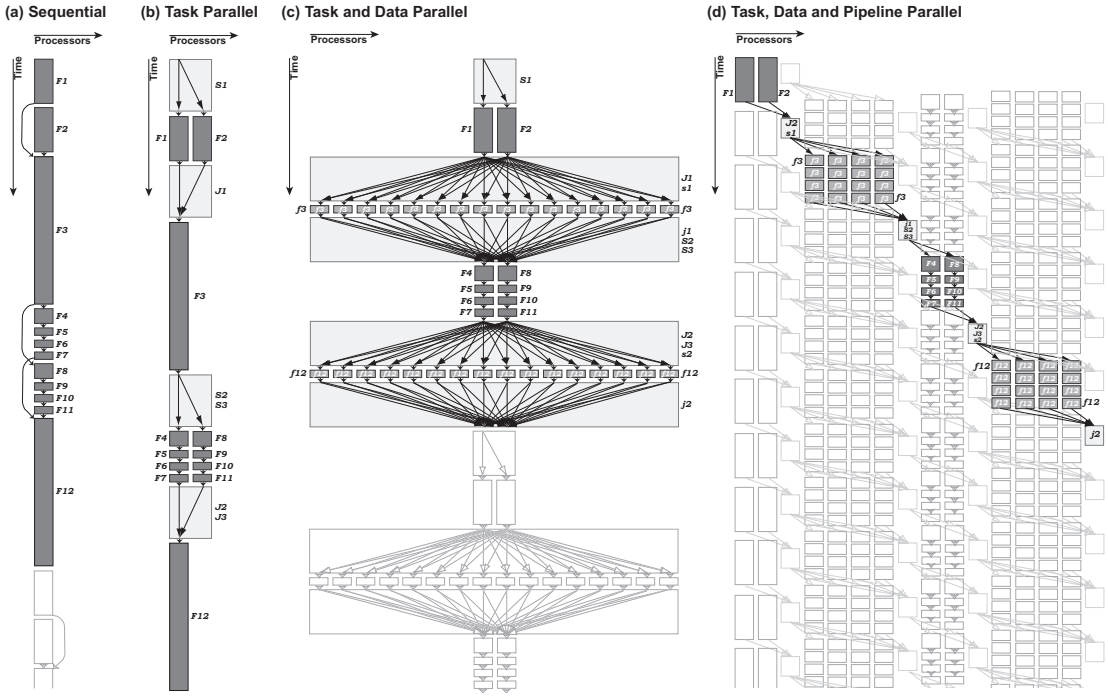


图 4.1　一个基于StreamIt语言的软件流水实例[14]

总的来说，该技术利用流语言特性在多核上实现了粗粒度的并行，并且在体系架构层次上具有良好的移植性。实际上，通过调整适应filter分裂的计算与通讯

的比值，可以更好地适应不同的体系结构。另外，对于一些基本的算法，如粗化粒度，智能分裂，划分，选择性的熔合等，在很大程度上是体系无关的。

4.3　线程级投机软件流水

该类软件流水算法与传统的软件流水算法有点类似，但它所针对是粗粒度并行。通过一些投机算法，及相应的通讯同步机制，抽取出线程，并把这些线程映射到多核中的不同核中执行。

论文[16]提供了一种线程级投机软件流水的编译技术，从程序式语言实现的并行或非并行的非完美循环，生成一种适应多核体系结构的多线程软件流水调度，并且同时保证必要的同步、非死锁和非缓冲区溢出，并且在保证原有指令级并行的条件下来实现线程级并行。

该编译技术首先使用了一维软件流水（Single-dimension Software Pipelining，SSP）技术来实现软件流水[17]。SSP是受限资源的软件流水方法，它针对的是单核体系结构中的完美和非完美的循环嵌套问题。SSP主要有两个优点: 支持嵌套循环中各层的软件流水；它能利用外层循环的ILP或数据局部性。其局限性在于，它对最外层循环的并行个数受限于有限的处理器资源。如图4.2。

然后通过一种称之为多线程SWP（Multi-Threaded SWP，MTS）的方法，把嵌套循环中的循环迭代分别映射到多核芯片的每一个核中，实现线程级并行。每个迭代循环初始化之间会有一定的间隔，而每个迭代的执行则利用了多核的并行实现。这样多线程的调度则与每个线程的调度无关，在实现了TLP的同时，保留了ILP。该方法通过共享内存实现通讯，实现了一个基于Lamport’s Clock的轻量级的同步机制，保证了代码的正确性，并实现一个砖瓦机制（Tiling Mechanism）保证了灵活性，最后保证结果调度无死锁。MST主要针对于三个目标：

- 解决依赖和资源限制问题;
- 实现一个轻量级同步机制;
- 解决外层循环的交叉迭代依赖

该编译技术实现了基于Lamport算法的轻量级同步机制。简单的说，Lamport算法又称面包房算法，是一种先来先服务算法。该算法跟很多银行采用的排队机制一样。客户到了银行，先领取一个服务号。一旦某个窗口出现空闲，拥有最小服务号的客户就可以去空闲窗口办理业务。

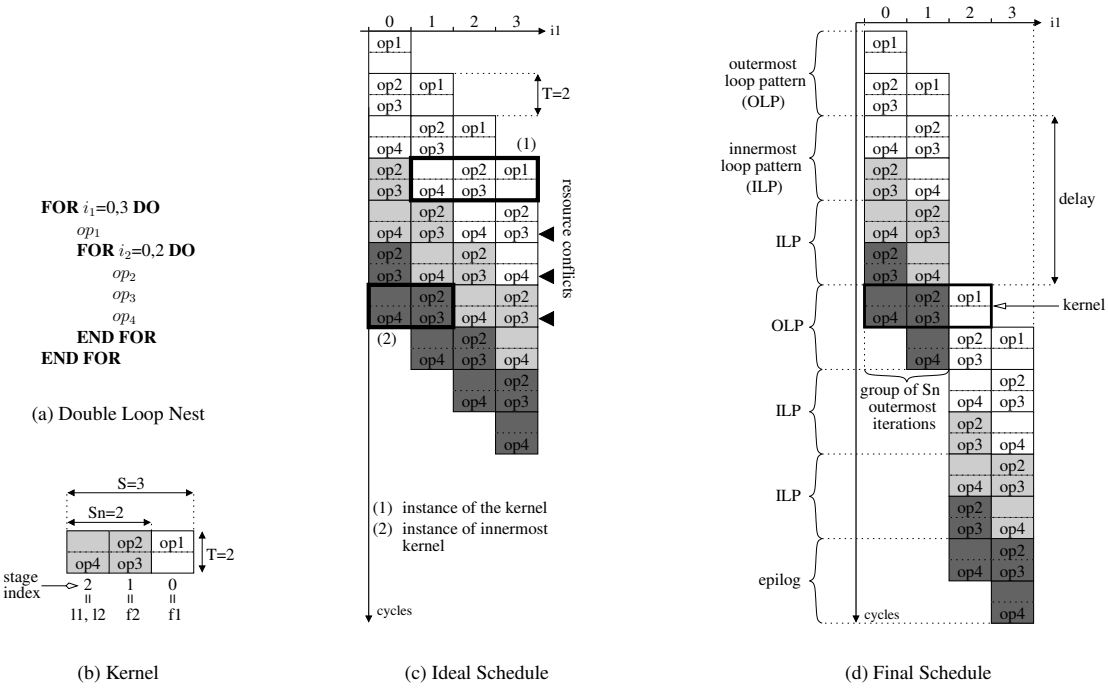


图 4.2　SSP例子[16]

更确切的，Lamport算法①利用事件排序方法，对要求访问临界资源的全部事件进行排序，并且按照先来先服务的原则，对事件进行处理。该算法规定[18]，每个进程Pi，在发送请求消息Request时，应该为它打上时间邮戳（Ti，i），其中Ti是进程Pi的逻辑时钟值，而且在每个进程中都保持一个请求队列，队列中包含了按逻辑时钟排序的请求消息。Lamport 算法用以下五项规则定义：

- 当进程Pi要求访问某个资源时，该进程将请求消息挂在自己的请求队列中，也发送一个Request（Ti，i）消息给所有其他进程。
- 当进程Pj收到Request（Ti，i）消息时，形成一个打上时间邮戳的Reply（Tj，j）消息，将它放在自己的请求队列中。应该说明，若进程Pj收到Request（Ti，i）消息前，也提出过对同一资源的访问请求，那么其时间邮戳应该比T（Ti，i）小。
- 若满足以下两个条件，则允许进程Pi访问该资源：Pi自身请求访问该资源的消息已经处于请求队列的最前面；Pi已经接收到从其他所有进程发回的

①. Leslie Lamport于1978年为分布式系统同步设计，Lamport同时是LaTeX的作者，仰慕!

响应消息，这些消息上的邮戳时间晚于T（Ti，i）。

- 为了释放该资源，Pi从自己的请求队列中消除请求消息，并发送一个打上时间邮戳的Release消息给其他所有进程。
- 消息后，从自己的队列中消除Pi的Request（Ti，i）消息。

这样，当每一个进程要访问一个共享资源时，本算法要求该进程发送3（N-1）个消息，其中（N-1）个Request消息，（N-1）个Reply消息，（N-1）个Release消息。

另处，需要注意的是，该编译方法假设所有的线程单元不共享寄存器。交叉迭代寄存器依赖将被转化成交叉迭代内存依赖，然后通过内存spill指令解决该依赖。而这里交叉迭代寄存器依赖是在编译期间发现的。

第 5 章 总结

多核技术已经成为当前处理器设计技术的主流，带来了巨大的机会和挑战。多核技术已经逐渐发展出多个研究方向，这些研究方向不断细化也不断交融，每个方向都对多核的应用有着不同程度的影响。

论文首先宏观讨论了当前多核技术的趋势和挑战，并且较为详细地讨论了其中的两块重要内容：核间通讯机制和多核自动化设计。然后，论文对多核技术相关的软件方法技术作了综述。要准确有效地利用多核的并行性等优势，相应的软件技术是必不可少的。多核上的软件方法的关键问题是如何把抽取出来的并行的执行单元映射到不同的核上。与之相关的，如何抽取、以及多核并发执行时的通讯同步机制也成为一个研究的重点。

多核技术的软件方法根据其目标，或其依赖的基础，分为细粒度并行软件流水和粗粒度并行软件流水。在讨论这两个软件流水技术的基础上，分别对当前国内外研究中典型的实现方法做了详细地分析。通过这些分析，论文试图宏观地呈现多核软件流水的现状，并努力把撑其本质。

参考文献

[1] Gordon E M. Cramming more components onto integrated circuits. Electronics, 1965

[2] Parkhurst J, Darringer J, and Grundmann B. From single core to multi-core: preparing for a new exponential. Proceedings of the 2006 IEEE/ACM international conference on ICCAD, 2006

[3] Barroso L, Gharachorloo K, McNamara R, et al. Piranha: A scalable architecture based on single-chip multiprocessing. ISCA-27, 2000

[4] Kumar R, Zyuban V, and Tullsen D. Interconnections in multi-core architectures: Understanding mechanisms, overheads and scaling. Proceedings of the 32nd Annual International Symposium on Computer Architecture, 2005

[5] Ottoni G, Rangan R, Stoler A, et al. Automatic thread extraction with decoupled software pipelining. Proceedings of the 38th IEEE/ACM International Symposium on Microarchitecture, 2005

[6] Hennessy J and Patterson D. Computer Architecture: A Quantitative Approach. China Machine Press, 3 edition, 2003

[7] Tzen T and Ni L. Dependence uniformization: a loop parallelization technique. Parallel and Distributed Systems, IEEE Transactions on, 1993

[8] Taylor M, Lee W, Amarasinghe S, et al. Scalar operand networks. IEEE Transactions on Parallel and Distributed Systems, 2005

[9] Smith J. Decoupled access/execute computer architectures. ACM Transactions on Computer Systems (TOCS), 1984

[10] Rangan R, Vachharajani N, Vachharajani M, et al. Decoupled software pipelining with the synchronization array. Parallel Architecture and Compilation Techniques, 2004

[11] Allen R and Kennedy K. Optimizing compilers for modern architectures: A dependence-based approach. 2002

[12] Garay M and Johnson D. Computers and intractability: A guide to the theory of np-completeness. WH Freenman, 1979

[13] Vachharajani N, Rangan R, Raman E, et al. Speculative decoupled software pipelining. Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques, 2007

[14] Gordon M, Thies W, and Amarasinghe S. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. Proceedings of the 12th international conference on ASPLOS, 2006

[15] Thies W, Karczmarek M, and Amarasinghe S. Streamlt: A language for streaming applications. Compiler Construction: 11th International Conference, Cc, France, 2002

[16] Douillet A and Gao G. Software-pipelining on multi-core architectures. Parallel Architecture and Compilation Techniques, 2007. PACT, 2007

[17] Rong H, Zhizhong N, Beijing C, et al. Single-dimension software pipelining for multi-dimensional loops. Code Gereration and Optimization, 2004, pages 163–174

[18] Lamport L. Time, clocks, and the ordering of events in a distributed system. Communications of the ACM, 1978