Prophet推测多线程系统设计与实现

李 钟 赵银亮 杜延宁
(西安交通大学计算机科学与技术系 西安 710049)

摘 要 推测多线程技术通过推测执行的方式开发应用程序的线程级并行性,以提高程序执行性能。该技术一般通过执行模型来检测运行时可能的线程推测错误情况,并采取合适的机制恢复程序正确运行。描述的Prophet是一种基于硬件实现的推测多线程执行模型。重点描述了Prophet执行模型针对执行模型设计的关键问题的解决方案,包括Prophet的线程状态控制和多版本的Cache系统,Prophet的多版本Cache系统提供了推测数据缓存功能,并使用基于总线监听的Cache协议实现了数据依赖违规检测。还给出了使用Olden基准程序对Prophet执行模型进行功能和性能测试的结果,并分析说明了Prophet系统可以有效地开发应用程序的线程级并行性。

关键词 推测多线程,线程级并行,推测多线程执行模型,推测多线程体系结构
中图法分类号 TP303 **文献标识码** A

Design and Implementation of the Prophet Speculative Multithreading System

LI Zhong ZHAO Yin-liang DU Yan-ning
(Department of Computer Science and Technology, Xi'an Jiaotong University, Xi'an 710049, China)

Abstract Speculative Multithreading(SpMT) improves the performance by means of exploiting speculative thread-level parallelism. In SpMT, Thread runs speculatively and SpMT execution model is used to detect and recover from misspeculation. This paper described Prophet, which is a hardware SpMT execution model implementation. Prophet execution model was described in details in this paper, including thread state control mechanism and Prophet Multi-versioning Cache system. Multi-versioning Cache system provides the speculative data buffering functionality and uses a snooping bus based cache coherence protocol to detect data dependence violation. This paper also presented the evaluation of Prophet excution model via Olden benchmark, which proves that Prophet system could archive significant speedup to non-numeric programs.

Keywords Speculative multithreading, Thread level parallelism, Speculative multithreading execution model, Speculative multithreading architecture

1 引言

发掘线程级并行性(Thread Level Parallelism, TLP)是程序并行性开发的一个可行方案。TLP的实现途径主要有同时多线程(Simultaneous Multithreading, SMT)和单芯片多处理(Chip Multiprocessor, CMP)两种。

在SMT和CMP体系结构提供的并行计算资源的基础上,可以采用自动并行化或手工并行化的方式对应用程序进行多线程化,然后通过线程的并行执行来提高程序的执行性能。传统的多线程模型中,编译器需要保证并行执行的线程之间的数据和控制无关性,这样的策略对数值型的规则的应用程序是有效的,然而对于非数值型的一般应用程序,因其程序结构不规则及程序代码间的控制和数据依赖关系复杂,编译器往往难以找到数据和控制无关的线程而导致线程划分失败。

推测多线程技术(Speculative Multithreading, SpMT)提出了推测的线程级并行化方案。在推测多线程中,应用程序同样被划分为多个线程,但不同的是,并行执行的线程之间可以存在数据和控制依赖关系,因此线程的执行是推测性的。推测多线程系统需要在运行时检测推测线程的执行情况,并在发生推测失败时采取合适的机制恢复程序执行,以保证程序运行的正确性。推测多线程编译器不需要保证线程间的数据和控制无关性,所以可以对原本难以进行线程化的程序进行线程划分。几乎所有的应用程序都能通过推测多线程技术来开发其线程级并行性。

在推测多线程技术中,推测多线程执行模型定义了推测多线程系统对线程推测执行所提供的运行时支持。推测多线程执行模型可以通过硬件方式或者软件方式来实现。本文将介绍Prophet推测多线程执行模型。Prophet执行模型使用硬件方式实现,采用了预计算方案来计算推测线程的live-in值,并使用多版本的Cache系统来实现数据缓存、live-in传递、依赖违规检测等功能。

到稿日期:2010-03-19 返修日期:2010-06-20 本文受国家高技术研究发展计划(863)(2008AA01Z136)资助。
李 钟(1984—),男,硕士,主要研究方向为推测多线程技术,E-mail:phenixsen@gmail.com;赵银亮(1960—),男,教授,博士生导师,主要研究方向为面向粒的编程模型、推测多线程技术;杜延宁(1978—),男,博士生,主要研究方向为推测多线程技术。

本文第2节将介绍推测多线程执行模型的一般研究内容;第3节是对Prophet编译器的简单描述;第4节介绍Prophet执行模型的设计和实现;第5节对Prophet的多版本Cache系统进行进一步描述;第6节是测试和分析;最后总结全文。

2 推测多线程执行模型概述

推测多线程技术的研究包括推测多线程编译器和执行模型两个方面。编译器使用激进的策略将串行程序划分为多个线程,但并不保证线程之间的数据和控制无关性。数据和控制依赖关系在运行时由执行模型解决,执行模型检测线程的推测执行情况,并采用合适的机制保证线程的正确执行。

2.1 执行模型的研究内容

执行模型研究的是保证线程推测执行正确和高效的方案。一般来说,执行模型的研究需要解决线程状态控制、推测数据缓存、live-in传递、数据依赖违规4个方面的问题。

线程状态控制:线程状态控制包括线程推测级判定以及线程发起、撤销和重启等操作的实现。在SpMT中,并行运行的线程之间存在逻辑上的先后关系,使用推测级别来描述这样的先后关系。线程的数据访问、数据提交等操作需要反映线程的逻辑顺序,以得到和串行执行时相同的结果。因此执行模型需要提供合适的推测多线程操作和状态转换机制,以保证程序的正确执行。

推测数据缓存:推测线程共享存储地址空间。执行模型需要保证只有非推测线程才能提交执行结果,否则会造成数据提交冲突而导致程序执行错误。执行模型一般通过提供推测数据的缓存方案来解决数据提交问题。当线程在推测状态执行时,写出的推测数据会被缓存,直到该线程成为非推测线程才被提交。

live-in传递:live-in指当前线程未定义而使用的数据,这样的数据需要在运行时从其它线程获取,以消除潜在的数据违规问题。按照程序的顺序执行语义,线程应该获取其前序线程对该变量的最新定义值,执行模型需要提供相应的机制向线程传递合适的live-in值。

数据依赖违规检测:即使采用了合适的机制进行live-in传递,数据依赖违规问题仍然存在。而发生数据依赖违规的原因在于线程之间对共享数据发生了WAR冲突。执行模型需要在执行写操作时进行检测操作,并在发生WAR冲突之后进行必要的恢复操作。

2.2 相关研究工作

推测多线程执行模型可以通过硬件或者软件的方式实现。硬件方案的研究主要关注体系结构的设计和实现,相关的研究包括Multiscalar[1],Hydra[2],STAMPede[3],SPSM[4],Mitosis[5,6]以及国内的SMA[7,8]和SWT[9]等。其中,MultiScalar使用ARB协议[10]来解决存储器数据依赖问题;Hydra则使用了特殊的Write Buffer来缓存推测数据,并使用基于冲突检测的写直达策略来解决线程间数据依赖问题。从整体结构上来说,这些研究一般都采用了两级Cache结构,使用L1 Cache来保存推测数据,并通过L1 Cache间的相关协议实现live-in传递和数据依赖违规检测。SpMT体系结构的一些技术已经得到广泛研究和认可。而另一方面,为了保证线程推测执行的效率,一些新的技术仍然在不断地被研究。文献[11]提出了几项提高SpMT系统性能的微处理器技术。而Mitosis[5,6]则提出在推测线程之前执行预计算片段(Pre-computation Slice,P-slice)来计算live-in的方案。除了硬件方式之外,还可以通过软件方式来实现推测多线程执行模型,相关的研究包括SableSpMT[12],jrpm[13],JavaSpMT[14]等。

3 Prophet编译器

Prophet编译器和Prophet执行模型是构成Prophet系统的两个部分。编译器负责进行程序分析和线程划分,而执行模型则提供对线程推测执行的运行时支持。Prophet编译器[15]以程序控制流图为基础,采用自上而下的方式对程序进行线程划分。在划分过程中,编译器主要考虑线程的粒度、线程间的数据和控制依赖3个因素。关于Prophet编译器的详细描述请参考文献[15]。

4 Prophet推测多线程执行模型

Prophet推测多线程执行模型是使用硬件方式实现的执行模型,该模型采用了Mitosis[6]提出的预计算技术,同时支持数据和控制推测。Prophet执行模型采用乱序的线程发起策略,并使用推测的多版本Cache系统实现live-in传递和数据依赖违规检测。Prophet目前并不考虑线程之间的寄存器依赖问题,而依靠编译器来保证线程之间的寄存器无关性。

4.1 Prophet线程模型

Prophet执行模型使用了Mitosis中提出的预计算概念。如图1所示,线程由线程发起点(Spawning Point,SP)和准控制无关点(Control Quasi-independent Point,CQIP)标识。CQIP是线程的间隔点,它既是前一个线程的结束点,也标识了下一个线程的开始。当线程执行到SP点时,会发起一个新的线程来执行CQIP点处的线程。推测线程被发起之后,它首先需要执行一小段预计算代码片段,来计算推测线程所需的live-in。预计算片段由编译器产生,它是线程发起点SP到线程开始点CQIP之间代码的简化版本。当执行完预计算之后,程序跳转到CQIP点之后执行线程体代码。在推测执行的过程中,线程会优先读取预计算得到的live-in值。当线程执行到下一个线程的CQIP点时,它停止执行并等待自己成为非推测的线程。之后,当前线程对其直接后继线程的预计算结果进行验证。如果预计算正确,那么当前线程提交其执行结果,后继线程成为非推测线程。否则,后继线程会被撤销,当前线程继续往下执行。

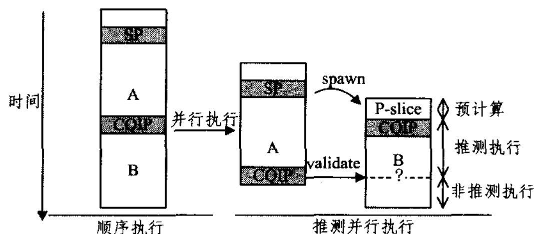

图1 Prophet线程模型

4.2 控制推测

Prophet执行模型支持控制推测。Prophet编译器可能会生成依赖于条件分支的线程发起操作,当发生这样的情况时,编译器会在另一分支处插入撤销该线程的指令,执行模型通过执行该条撤销指令而撤销错误发起的线程,以保证程序

执行的正确性。

4.3 乱序发起和线程推测级

Prophet 执行模型采用了文献[11]中提出的乱序发起策略。该策略规定，如果一个线程体内有多次线程发起操作，那么线程的发起顺序应该与被发起线程的逻辑顺序相反，也就是说逻辑顺序上处于最后的线程总是被最先发起。这样的策略保证可以在线程体内进行多次发起操作，比顺序发起策略更为高效[11]。

线程的推测级在其运行时具有重要的作用。当采用乱序发起策略时，线程的激发顺序与其逻辑顺序并不一致，因此不能简单地依靠线程的发起时机来判定线程的推测级。文献[11]提出使用分裂时间戳(Splitting Timestamp Interval, STI)方案来解决线程的推测级判定问题。

分裂时间戳方法的实现基于这样的事实：每当线程发起时，子线程总是会成为父线程的直接后继线程。这是因为，首先，子线程的推测级别必然高于父线程；再者，线程发起策略规定当线程进行多次发起时，总是会首先发起推测级别较高的线程。因此在线程发起时，必然不存在推测级处于待发起线程和父线程之间的线程正在运行，因此待发起线程总是会成为父线程的直接后继线程。基于这样的认识，Prophet 使用推测级链表来维护线程的推测级别。该链表的每一个节点表示当前运行的线程，并以推测级顺序链接。当产生发起操作时，新发起线程被插入到父线程之后，如此便可以正确维护线程的推测级关系。另外，为了能够快速比较线程的推测级而无需遍历链表，链表中的每一个节点都被赋予相应的推测级值，并在链表发生变更时进行更新。

4.4 Prophet 体系结构

Prophet 体系结构是 Prophet 执行模型的硬件实现，该结构具有一个类似于 Hydra CMP 的多核结构，如图 2 所示。Prophet 结构由处在一个芯片上的多个线程单元组成，每一个线程单元是一个 RISC 核。

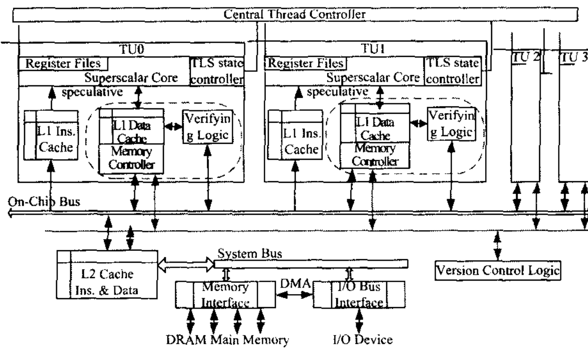

图 2 Prohpet 微处理器结构

Prophet 线程单元是可以进行独立计算的功能单元，它拥有独立的寄存器集、算术逻辑单元、控制单元以及控制线程推测状态的 SpMT 控制单元。线程单元拥有私有的 L1 数据 Cache 和 L1 指令 Cache。L2 Cache 则由所有的线程单元共享。私有的 L1 数据 Cache 用于存放线程单元的推测数据，当线程处于推测执行阶段时，数据只能写出到 L1 Cache，线程执行到非推测阶段时，才可以向 L2 Cache 提交执行结果。

线程单元之间通过总线连接。Prophet 体系结构使用特殊的版本控制逻辑(Version Control Logic, VCL)来实现多版本的 Cache 系统，以解决 live-in 传递、数据依赖违规等问题。线程单元内部的验证逻辑(Verifying Logic)用于进行 live-in 值验证，验证逻辑通过总线获取其直接后继线程的预计算结果。

4.5 Prophet 线程状态控制

因为引入了预计算技术，Prophet 执行模型推测运行时的状态变化较为复杂，图 3 展示了 Prophet 线程单元执行时的状态转换图。线程状态变化受线程控制指令驱动，线程控制器负责进行全局的线程操作。

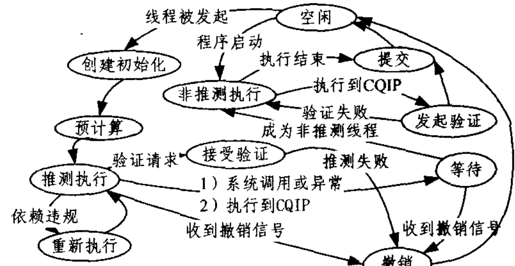

图 3 线程单元状态转换图

当线程执行发起操作时，它需要向线程控制器请求空闲的线程单元。如果当前没有空闲的线程单元，那么线程控制器会采取低优先级优先的算法来重新分配计算资源，即如果当前存在推测级高于待发起线程的线程，那么其中推测级最高的线程将会被撤销，从而产生空闲的线程单元来执行待发起线程。如果通过这样的策略还是无法找到空闲的线程单元，那么线程发起失败，但这并不会影响线程的正确执行。

如果找到空闲的线程单元，线程控制器会对其进行初始化操作，随后该线程单元开始预计算片段的执行。线程在预计算片段的最后通过跳转指令跳转到线程体正文，开始正式的推测执行过程。

当线程执行到 CQIP 点时，它需要等待自己成为非推测线程，然后对其直接后继线程预计算的 live-in 值进行验证。如果验证正确，那么当前线程提交其数据，并结束运行，其直接后继线程成为非推测线程。否则，其直接后继线程及以后的所有线程将会被撤销，执行结果将被丢弃，线程单元转入空闲状态，而当前线程则继续以非推测方式执行 CQIP 之后的代码。

Prophet 在线程推测执行的过程中检测线程的数据依赖违规情况。当发生数据依赖违规时，违规的线程之后的所有线程将被撤销，而违规线程则会被重新执行。

当推测线程执行过程中发生系统调用或者异常时，因为这些操作不能以推测方式执行，所以当前线程需要进入等待状态，直到线程成为非推测时才可以进行相应的操作。

5 多版本 Cache 系统

在硬件实现的 SpMT 系统中，基本上都使用 Cache 系统来解决推测数据缓存、live-in 传递和数据依赖违规检测等问题，Prophet 采用了相同的方案。Prophet Cache 系统使用了类似于 SVC[16] 的结构和协议，但因为引入预计算技术，Prophet 系统不仅需要在多个线程单元之间存储同一位置的不同版本数据，而且线程单元内部也会有不同版本的数据。线程内的数据版本之间使用专门的版本号位进行区分。

本节将具体介绍 Prophet Cache 系统的设计方案。本节刚开始的部分将会给出一个基本的 Cache 系统设计方案，在

基本的方案里，认为 Cache 行长度为一个字长，并假设线程单元的操作都以一个字为单位。本节的最后部分将说明在 Cache 行大小超过一个字长和线程单元进行不同粒度的访问操作时的扩展方案。

5.1 L1 Cache 行设计

Prophet 为其 L1 Cache 中的 Cache 行添加了推测相关的域，图 4 给出了 Prophet L1 Cache 行的各个域的内容。

Ver	ID	V	M	RL	RR	Old	Address Tag	Data

Ver: Version Flag　　ID: Source Processor ID
V : Valid Flag　　M : Modify Flag
RL : Remote Loaded Flag　　RR: Remote Read Flag
Old: Old Flag

图 4　Cache 行内容

其中，Ver 表示当前 Cache 行的版本号。在线程运行的不同阶段，线程单元写出的数据被赋予不同的版本号。当线程在执行预计算时，写出数据的版本号为 0；当线程离开预计算时，版本号变为 1。随后，每当线程发起一个子线程，写出数据的版本号都加 1。

ID 为当前 Cache 行的数据来源线程单元的编号。当线程单元执行 Load 操作从其它线程单元获取数据时，ID 被置为数据来源的线程单元编号。而当线程从 L2 Cache 获取稳定的数据时，使用特殊的 STABLE_ID 表示。

M 为修改位。如果线程对该 Cache 行进行了修改，那么该位被置为 1。

RL 为远程装载位。该位为 1，表示数据是从其它线程单元载入的，线程单元对该 Cache 进行了未定义读取操作。

RR 为远程读取位。如果 Cache 行被其它线程读取，那么该位被设置为 1。

Old 位为 1，表示该 Cache 行不是当前线程内的最新版本。当线程提交时，只需要提交非 Old 的 Cache 行即可。

5.2 基于总线监听的 Cache 一致性协议

Prophet 采用基于总线监听的 Cache 一致性协议来处理 Cache 操作。当发生 Load 操作时，如果对线程单元内部的 L1 Cache 访问命中，则只需要读取私有 L1 Cache 中的数据即可。相反地，如果没有命中，那么线程单元会发送总线请求，通过版本控制逻辑向其它线程单元请求 Cache 行数据的最新定义版本；当发生 Store 操作时，Prophet 需要向其它线程广播该写出操作，以检测可能的数据依赖情况；当发生其它的一些操作时，Cache 系统同样可能会通过总线向其它线程传递信息。

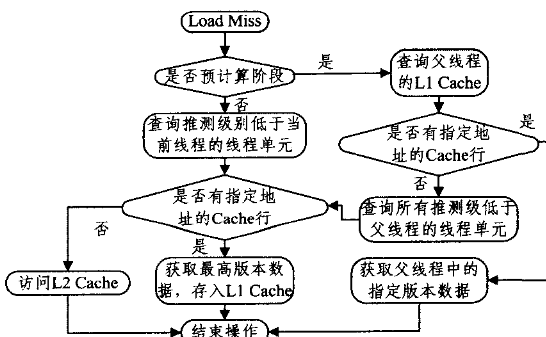

图 5　LoadMiss 操作流程图

Load 操作：当线程单元发生 Load 操作时，线程单元首先会尝试载入私有 L1 Cache 中该位置数据的最新版本。因为总是首先尝试载入 L1 Cache 中的数据，所以可以保证预计算片段计算的 live-in 值总是会被首先读取。如果访问 L1 Cache 失败，则发生了 Load Miss，线程单元发送 BusRead 总线信号，通过 VCL 来获取合适的数据。LoadMiss 时的基本操作如图 5 所示，根据当前线程是否处于预计算阶段，来确定需要采取的操作。

如果当前线程并非处于预计算阶段，那么当前线程应该看到的是 CQIP 点之前的机器状态。线程单元会发出 BusRead 信号，向优先级低于自己的线程单元发起读请求，并通过版本控制器按推测级顺序逆向查找所需要的 Cache 行。如果找到这样的 Cache 行，则线程单元将 Cache 行存入自己的 L1 Cache 中，并设置该 Cache 行的 ID，RL 等域。而源 Cache 行则需要设置 RR 位为 1，表示该 Cache 行被读取。如果在推测级较低的线程单元中找不到所需的 Cache 行，那么线程单元需要请求 L2 Cache 获取该数据的稳定版本。

如果线程处于预计算状态，那么线程应该看到的是 SP 点之前的机器状态，这时父线程和子线程实际上执行的是同一段代码的不同版本。为了保证父线程的写出操作不影响子线程的执行预计算，Prophet 使用版本号来区分 SP 点前后的数据版本。当线程发起子线程时，写出操作的数据版本号发生了变化，在发起时，父线程会传递发起点的数据版本号给子线程，当子线程处于预计算状态时，它会根据保存的数据版本号来读取 SP 点之前写出的数据。

Store 操作：当发生 Store 操作时，线程单元会根据当前的版本计数写出某个版本的数据。如果在前一次对该地址进行写出操作之后，版本计数并没有变化，那么 Store 操作其实是对现有 Cache 行的修改，否则线程单元需要写出新的 Cache 行。无论是哪种情况，当执行完数据的写出操作之后，线程单元的 M 位都会被设置为 1，并且 RR 位置为 0，表示写出操作之后，该 Cache 行没有被其它线程读取。另外，当写出新的 Cache 行之后，原先的 Cache 行会被置 Old 位，表示该 Cache 行已经不再是最新的版本。

如果写出操作是该线程单元对 Cache 行的第一次写出，或者在前一次写出之后，该 Cache 行被其它线程单元读取，即 Cache 行的 RR 位为 1，那么当前的写出操作可能会产生 WAR 冲突，线程需要向推测级高于自己的线程发出 BusWrite 信号，以进行数据依赖违规检测。

当线程单元收到来自其它线程的 BusWrite 信号时，它会检查是否发生了 WAR 冲突。如果当前线程曾经从其它线程载入了该 Cache 行，并且载入的 Cache 行的优先级低于当前写出的数据版本，那么说明发生了 WAR 冲突。线程单元从其它线程单元载入数据时，会保存源线程单元的 ID。当进行数据依赖冲突检测时，线程单元根据该 ID 号来判断载入的数据和当前写出操作的数据的推测级。只有当写出操作的推测级等于或高于原先载入数据的推测级时，才认为发生了数据依赖冲突。若确定发生数据依赖冲突，那么当前线程的所有后继线程需要被撤销，而当前线程则需要被重启。

Validate 操作：当发生 Validate 操作时，当前线程向其直接后继线程发送 Validate 信号，获取其预计算的 live-in 值，并进行相应的检测操作。在执行预计算时，线程单元的数据版本为 0，因此 live-in Cache 行必然具有特征($M=1 \land Ver=0$)。

Commit 操作：在进行 Commit 操作时，线程只需要提交数据的最新定义版本，也就是 L1 Cache 中的所有非 Old

Cache 行。另外，当线程完成 Commit 操作之后，需要发送信号通知其后继线程自己已经完成提交操作，后继线程需要更新自己 L1 Cache 中对应的线程单元 ID 值为 STABLE_ID，以表示这些载入的数据都已经是稳定的。

5.3 预计算

在预计算阶段，线程单元需要做一些额外的操作。首先，预计算载入的数据不能被其它线程读取，因此在预计算阶段发生 Load 操作时，被载入的 Cache 行总会被置为 Old。另外，当线程进入正式的推测执行时，当前线程不能再读取预计算时载入的值，因此需要在离开预计算片段时将这些预计算阶段载入的非 live-in 值置为无效。

另一个需要考虑的问题是，预计算的结果是否应该被推测级较低的线程看到？如果预计算失败，那么推测级低于当前线程的线程也会被撤销。而编译器产生预计算片段时，会采取精简和优化的策略，不会产生不必要的 live-in。因此在 Prophet 中，当线程向其前序线程请求数据时，同样会访问该线程预计算结果。

5.4 针对 Cache 行长度和访问粒度的扩展

在扩展方案中，Cache 行的 M 和 RL 标记需要被扩展为多个位，其中每一位对应 Cache 行数据的一个字节。当发生 Load 和 Store 操作时，线程单元对 M 和 RL 域进行细粒度的设置。

扩展方案中，Cache 系统基本操作单位是 Cache 行。当执行 Load 或 Store 操作时，如果当前线程内部没有指定地址标记的 Cache 行，那么称发生 AccessMiss，线程单元需要向其前序线程请求该 Cache 行的最新定义版本，之后才能对该最新定义版本进行读取和写出。

在扩展方案中，预计算结果能只是 Cache 行中的几个字节，而其它字节是无效的。因此当读取 live-in 时，还需要向前序的线程请求 Cache 行，然后将 live-in 合并入载入的 Cache 行中。

6 测试和分析

本文选择了 Olden 基准程序包作为测试用例，Olden 基准程序大都具有复杂的控制流结构，并且使用了大量的指针运算，数据依赖关系复杂。选择 Olden 基准程序能很好地证明本文所提方案对非数值型应用程序的并行化能力。

试验所采用的模拟器是在 SUIF 中间表示基础上开发的 ProphetVM 虚拟机。该虚拟机以线程化的中间表示为输入进行解释执行，以程序驱动方式对 Prophet 进行动态模拟。ProphetVM 中并没有寄存器概念，它以指令周期为单位模拟线程单元的并行执行，并实现了推测的 Cache 系统，以模拟 Cache 的相关操作。ProphetVM 为中间表示指令规定了相应的指令周期，并通过统计程序执行的动态指令周期的方式来评价程序性能。

表 1 所列为 ProhpetVM 的默认指令开销设定。推测指令的开销都为 5 个指令周期，这是在考虑了 ProphetVM 推测执行的特性之后得出的估计值。ProphetVM 中没有寄存器，每一条 SUIF 指令都大概需要 3 次访存操作。因此一条 SUIF 指令的开销大概相当于 4 条 RISC 指令，推测指令的实际开销相当于 20 条 RISC 指令，这应该是足够了。

表 1 ProphetVM 指令开销

指令	开销(指令周期)	描述
spawn	5	线程发起指令
cqip	5	线程开始/结束指令
cancel	5	线程撤销指令
mul	2	乘法指令
div	2	除法指令
rem	2	取模指令
其它	1	其它所有指令

表 2 给出了基准程序在默认参数(即 4 个线程单元)时 ProphetVM 虚拟执行的性能参数统计。从统计参数可以看出，对基准程序的推测执行基本可以达到 1.3 的加速比，说明推测执行模型的确可以提高程序的执行性能。但从表中可以看出，在虚拟执行时，线程的撤销概率仍然比较大。这是因为 ProphetVM 并没有寄存器模拟功能，因此 SUIF 指令的操作数存取都需要访存，这必然会产生不必要的数据依赖违规而导致线程的撤销概率增加。

表 2 默认参数下的基准程序性能参数

基准程序	非推测执行周期	推测执行周期	发起线程数	被撤销线程数	加速比
MST	2467402	1678504	30796	7314	1.47
Power	1145160	867287	22471	5498	1.32
Treeadd	245741	204532	8132	3524	1.20
TSP	1150388	881124	24329	8932	1.30
Voronoi	364091	271421	6432	2548	1.34

图 6 给出了线程单元数为 2,4,8 时的推测加速比。可以看出，当线程单元数增加时，程序执行的加速比会有相应的提高。

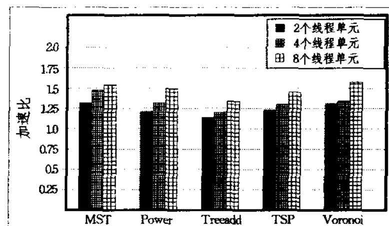

图 6 不同线程单元数情况下的加速比情况

结束语 本文介绍了 Prophet 执行模型的设计和实现方案。该模型采用了预计算技术，同时支持数据和控制推测。Prophet 执行模型采用乱序的线程发起策略，并使用推测的多版本 Cache 系统实现 live-in 传递和数据依赖违规检测。通过 Olden 基准程序对 Prophet 执行模型的测试，可以得出 Prophet 执行模型功能正确的结论。同时整个 Prophet 推测多线程系统可以有效地开发应用程序，特别是非数值型应用程序的线程级并行性。

参考文献

[1] Sohi G S, Breach S E, Vijaykumar T N. Multiscalar Processors [C] // Proceedings of 25 Years ISCA: Retrospectives and Reprints. 1998: 521-532

[2] Hammond L, Hubbert B A, Siu M, et al. The Stanford Hydra CMP[J]. IEEE Micro, 2000, 20(2): 71-84

[3] Steffan J G, Colohan C B, Zhai A, et al. A scalable approach to thread-level speculation[C] // Proceedings of the 27th Annual International Symposium on Computer Architecture. New York, NY, USA: ACM Press, 2000: 1-12

[4] Dubey P K, O'Brien K, O'Brien K M, et al. Single-program

speculative multithreading (SPSM) architecture: compiler-assisted fine-grained multithreading[C]// Proceedings of the IFIP WG10.3 Working Conference on Parallel Architectures and Compilation Techniques. Manchester, UK: IFIP Working Group on Algol, 1995: 109-121

[5] Madriles C, Sánchez C G Q J, Marcuello P, et al. The Mitosis Speculative Multithreaded Architecture[C]// Proceedings of Parallel Computing: Current & Future Issues of High-End Computing. 2006: 27-38

[6] Quiñones C G, Madriles C, Sánchez J, et al. Mitosis compiler: an infrastructure for speculative threading based on pre-computation slices[C]// Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation. New York, NY, USA: ACM Press, 2005: 269-279

[7] 肖刚，周兴铭，徐明，等. SMA：前瞻性多线程体系结构[J]. 1999, 22(6): 582-590

[8] 邓鹍. 前瞻多线程编译优化技术的研究与实现[D]. 长沙：国防科技大学，2000

[9] 鲁建壮，王志英，张春元. 面向 SCMP 的多线程前瞻控制分析与设计[J]. 计算机工程与科学，2006, 28(10): 128-130

[10] Franklin M, Sohi G S. ARB: A Hardware Mechanism for Dynamic Reordering of Memory References[J]. IEEE Transactions on Computers, 1996, 45(5): 552-571

[11] Renau J, Tuck J, Liu W, et al. Tasking with out-of-order spawn in TLS chip multiprocessors: microarchitecture and compilation [C]// Proceedings of the 19th Annual Iinternational Conference on Supercomputing. NY, USA: ACM Press, 2005: 179-188

[12] Pickett C J F, Verbrugge C. SableSpMT: A Software Framework for Analysing Speculative Multithreading in Java[C]// Proceedings of PASTE'05: Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering. New York, NY, USA: ACM Press, 2005: 59-66

[13] Chen M, Olukotun K. The Jrpm system for dynamically parallelizing java programs[C]// Proceedings. 30th Annual International Symposium on. 2003: 434-445

[14] Kazi I H, Lilja D J. JavaSpMT: A Speculative Thread Pipelining Parallelization Model for Java Programs[C]// Proceedings of the 14th International Parallel and Distributed Processing Symposium. 2000: 559-564

[15] 冯威. 基于 CMP 的推测多线程执行机制及其编译关键技术研究[Z]. 2007

[16] Vijaykumar T N, Gopal S, Smith J E, et al. Speculative Versioning Cache[J]. IEEE Trans. Parallel Distrib. Syst., 2001, 12(12): 1305-1317

[17] Olden Benchmark Suit[EB/OL]. http://www.cs.princeton.edu/~mcc/olden.html

(上接第 287 页)

何变换、投影变换采用硬件加速器形式，光照变换和纹理映射采用可编程处理器结构。我们分析了不少图形处理的染色程序和 DSP 图像处理程序，并按照分析的结果设计了一个扩展的 VLIW 处理机。这个机器的基本结构如图 8 所示。该机器包括 8 个向量处理器和 3 个标量处理器。每个向量处理器的宽度是 16 个标量数字。在 8 个向量处理器中：3 个是通用处理器，用于实现大多数算数和逻辑运算；5 个是专用处理器。其中，VMove 用来在向量寄存器间移动数据；Reduction 用来做算数缩减运算；BSort 用来做整数排序；VMult 是向量乘法器；Select/Spread/Generate 用来从向量中选取任意单元，将一个标量值用某个函数分布到向量中去，或者按照规定函数产生一个向量。

图 8　可编程处理器结构

3 个标量处理器中，一个是简单处理器，只能处理简单算数和逻辑运算；另一个的指令较多，还可以处理自动循环和转移；第三个可以执行整数除法和开平方。在这些处理器之间有数据传输网络，还有一组寄存器。处理器之间不做相关性检查，不做冒险处理，也没有 Inter-lock 机制。这些都由软件来完成。光照染色和纹理映射都用此处理器编程来实现。

结束语　本文论述了图形处理器流水线的设计思想。在提出了一条实用的图形处理流水线的基础上，研究了光照和纹理的基本原理，并进行了软件仿真和硬件设计。通过软件仿真设计结果进一步说明了我们设计的正确性。

图形处理器历经二三十年的发展，已经广泛应用于图形图像处理、视觉计算，甚至包括通用计算、大规模并行计算等等。目前图形处理器设计技术仍然掌握在国外两三家公司，可以查阅的资料较少。国内也有少量相似研究图形关键算法和硬件实现[9]，但核心技术仍受制于人。我们在这方面做了一定的研究，希望能对国内相关工作提供一些参考和借鉴。

参考文献

[1] Foley J D, van Dam A, Feiner S K, et al. Computer Graphics: Principles and Practice in C (2 edition) [M]. Addison-Wesley Professional, August 1995

[2] Phong B T. Illumination for Computer Generated Images [J]. CACM, 1975, 18: 311-317

[3] Blythe D. The Direct 3 D 1 0 system [J]. ACM Transactions on Graphics (TOG), 2006, 25(3): 724-734

[4] 吴恩华，柳有权. 基于图形处理器(GPU)的通用计算[J]. 计算机辅助设计于图形学报，16(5): 601-612

[5] Common Shader Core (DirectX HLSL). Microsoft[OL]. http://msdn.microsoft.com/en-us/library/bb509580(VS.85).aspx

[6] Gouraud. Continuous Shading of Curved Surfaces [J]. IEEE Trans. Computers, 1971, 20: 623-629

[7] Crow. The Aliasing Problem in Computer Generated Shaded Images[J]. CACM, 1977, 20(11): 799- 805

[8] Shreiner D，等. OpenGL 编程指南(第 6 版)[M]. 北京：机械工业出版社，2009

[9] 杨毅. 面向移动设备的真实感图形处理系统设计与实现[D]. 合肥：中国科技大学，2008

[10] 邱航，陈雷霆. 基于点的计算机图形学研究与进展[J]. 计算机科学，2009, 36(6): 10