文章编号: 1001- 2486(2008) 01- 0037- 05

流编程模型下的存储一致性模型*

邓 宇[1], 杨学军[1], 戴华东[1], 王 勐[2]
(1. 国防科技大学 计算机学院, 湖南 长沙 410073; 2. 总后勤部后勤科学研究所, 北京 100071)

摘 要: 在流编程模型下建立了一个新的存储一致性模型——流一致性模型, 它比传统的释放一致性模型更加松弛。讨论了流一致性模型对程序设计和系统设计的要求, 给出了一个正确的系统实现, 并且指出流一致性模型的编程和实现并不比现有的一致性模型复杂。

关键词: 流计算; 流编程模型; 存储一致性模型
中图分类号:TP302 文献标识码: A

Memory Consistency Model for Stream Programming

DENG Yu[1], YANG Xue-jun[1], DAI Hua-dong[1], WANG Meng[2]
(1. College of Computer, National Univ. of Defense Technology, Changsha 410073, China;
2. Institute of Logistics Sciences, the PLA General Logistics Department, Beijing 100071, China)

Abstract: A new programming model-the stream programming model is presented on the basis of the stream computing model. A new memory consistency model, named stream consistency for the stream programming model is proposed. In comparison with the traditional release consistency, the stream consistency is more relaxed. The requirements of the stream consistency for the program and system designs are discussed and a correct implementation is presented. The programmability and system overhead of the Stream Consistency is analyzed.

Key words: stream computing; stream programming model; memory consistency model

在现代共享存储计算机系统中, 随着系统规模的扩大, 数据一致性的维护对系统性能的影响越来越大。简单高效的存储一致性模型一直是人们追求的目标, 为此人们提出了多种存储一致性模型, 如顺序一致性[1]、处理机一致性[2]、弱一致性[3]以及释放一致性[4]等。一般而言, 越松弛的一致性模型, 其性能越好, 但对程序员的要求越高, 系统实现也越复杂。存储一致性模型与程序的访存行为密切相关, 而程序的访存行为决定于计算模型和编程模型。因此, 一旦计算模型和编程模型发生变化, 就可能产生新的存储一致性模型。

流编程模型是一种新的编程模型, 它对数据组织和数据交换有特殊的要求, 其程序的访存行为比传统的编程模型下的程序更加规则。对应这种变化, 流编程模型下的存储一致性模型应该比传统的释放一致性模型更加松弛。

1 流编程模型

在多媒体等应用中, 往往有大量的数据需要经过相同或相似的计算过程进行处理。针对这种特点, 流编程模型将计算过程划分成若干"核心"(Kernel), 每个核心的输入是一个由操作相同的数据构成的数据序列, 核心对输入的数据序列进行处理, 产生新的数据序列作为下一个核心的输入。这些数据序列就是"流"(Stream), 若干个核心通过流连接在一起就构成了一个流程序[5]。

流程序的访存行为具有明显的规律性, 即只在核处理数据之前批量读入数据, 然后在核处理完数据之后批量写回数据, 也就是只有整个流程序的输入流和输出流需要访问存储器, 其他时候并不访问存储

* 收稿日期:2007- 09- 18
基金项目:国家自然科学基金资助项目(60621003;60633050)
作者简介: 邓宇(1977—), 男, 博士生。

器。存储一致性模型与程序的访存行为密切相关,访存行为的简化和规则化将使得数据一致性的维护变得更加简单和容易,从而获得更加松弛的一致性模型。因此,在流编程模型下,流并行程序的存储一致性模型将比传统并行程序的释放一致性(Release Consistency,简称RC)等模型更加松弛。

基于以上对流编程模型的分析,我们从访存的角度对流程序进行定义。

定义1 流程序是一个三元组< streamload, kernel, streamstore> ,其中,streamload是输入流读取操作的集合,kernel是流程序内部所有核的集合,streamstore是输出流写回操作的集合。流的并行程序由多个流程序构成,它们之间通过流交换数据。

在下面的讨论中将流的读写看成一个整体,分别记为流读(记为sl)和流写(记为ss)。

2 流一致性模型的提出

现代存储一致性模型都是通过同步操作来维护数据一致性[6]。例如在RC模型中,同步操作分为获取操作acquire和释放操作release。acquire用于获取对共享存储单元的访问权,release则用于释放这种访问权。同样使用acquire和release这样的同步操作来维护流的存储一致性,但由于流程序访存行为的特殊性,对访存次序的限制也有所不同。

所谓一致性的维护,就是解决"冲突访问"[7]的访存次序问题。因此需要对流程序的冲突访问进行定义。

定义2 如果两个流访存操作所作用的数据流中有相同地址的元素,且至少有一个是流写操作,则称这两个操作为冲突的流访问操作。对应的数据流称为共享流。

根据定义2,流程序中可能存在以下3类冲突的流访问,分别是先读后写(sl- > ss),先写后读(ss- > sl),写后写(ss- > ss)。首先,对于sl- > ss型的冲突访问,意味着某个核要从某个地址开始取流,而之后另外一个核将向这个地址写入新的流,这种情况可以通过重新分配物理地址的方法加以化解。其次,对于ss- > ss型的冲突访问,意味着有多个核的输出流将作为某个核的输入流,在流编程模型里,这种情况是不被允许的。因此,需要解决的只有ss- > sl型的冲突访问,只需要在ss和sl之间加上同步,就能保证它们的访存顺序。另外,从数据一致性维护的角度看,一条数据流内的数据的地位是相同的,可以将同步操作的作用域限制在所关联的共享流上。

定义3 在一条共享流的读取(sl)之前有且仅有一个acq操作,其作用域仅限于这条共享流的读取,称为与之关联的acq操作;在一条共享流的存储(ss)之后有且仅有一个rel操作,其作用域仅限于这条共享流的存储,称为与之关联的rel操作。

定义4 synchron是同步操作的集合。RA是synchron上的一个二元关系,RA= {(rel, acq) | rel和acq与同一条共享流相关联},RA中的每个元素(rel, acq)称为同一组同步操作。

由此我们给出流的存储一致性模型,简称流一致性模型对访存事件发生次序的限制条件:

(1)必须且仅需保持同一组(rel, acq)操作的顺序;

(2)在任何共享流的读取开始之前,与其相关联的acq操作必须完成;

(3)在任何rel操作执行之前,与其相关联的共享流的写回必须完成。

操作完成是指该操作所产生的效果对所有的处理机可见。流的读取开始之前是指读入第一个流元素之前;流的写回完成是指流的最后一个元素写回完成。同一组(rel, acq)操作的含义由定义4给出。

图1比较了流一致性模型和RC模型对访存事件发生次序的限制,sl相当于r,ss相当于w,分别表示读和写。从中可以看出,相对RC模型,流一致性模型对访存次序限制的松弛主要表现在两个方面:首先,同步操作之间仅需维持rel_i- > acq_i的顺序一致性,下标i表示这是同一组同步操作;其次,同步操作与访存操作之间,仅需维护acq- > sl和ss- > rel的顺序。

3 流一致性模型的设计要求

存储一致性模型作为编程模型的一部分,是程序员与体系结构设计者之间的一个界面,它应该给出正确程序设计和正确结构设计的标准。下面分别讨论在流一致性模型中,程序正确性设计的标准和系

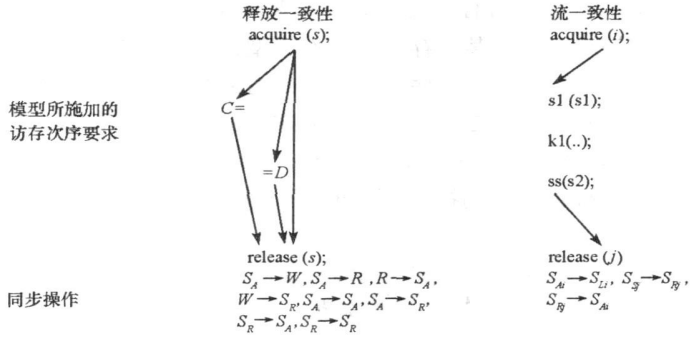

图 1　流一致性与释放一致性的比较

Fig. 1　Comparisons between stream consistency and release consistency

统正确性设计的标准。

任何程序都可以看作是由一个操作集合以及该操作集合上的一个顺序关系组成的[8]。由于存储一致性模型的视角是在处理机之下、存储器之上, 所以只需考虑访存操作之间的顺序关系。在流程序里面, 访存操作包括 sl, ss 以及同步操作。因此, 从序关系来看, 可以这样定义流程序:

定义 5　流程序 P 是一个序结构$< V(P), PO(P) >$, 其中 $V(P) = streamload \cup streamstore \cup synchron$, $PO(P)$ 是 $V(P)$ 上的一个全序关系。

由 N 个流程序 P_1, P_2, …, P_N 组成的流并行程序 $PRG(P_1, P_2, \cdots, P_N)$ 是一个序结构$< V(PRG), PO(PRG) >$, 其中, $V(PRG) = V(P_1) \cup V(P_2) \cup \ldots \cup V(P_N)$, $PO(PRG) = PO(P_1) \cup PO(P_2) \cup \ldots \cup PO(P_N)$ 是 PRG 的程序序。

从存储一致性的角度考来看, 在并行程序的执行过程中, 冲突访问的执行次序决定了执行结果[8]。另一方面, 冲突访问的执行次序又是由同步操作的执行次序决定的。因此, 同步操作的执行次序决定了并行程序的执行结果。我们将同步操作的执行次序称为程序的一个执行(execution)。

定义 6　流并行程序 PRG 在流一致性模型 S 中的一个执行 $ES(PRG) = \{(rel, acq) \mid (rel, acq) \in RA\}$。

定义 7　流一致性模型 S 中的一个执行 $ES(PRG)$ 的同步序, 是在该执行下被定序的冲突的流访问操作对的集合, 记为 $SO_S(E_S(PRG))$。即 $SO_S(E_S(PRG)) = \{(ss, sl) \mid ss \xrightarrow{PO} rel \xrightarrow{E_S} acq \xrightarrow{PO} sl\}$。

实际上, 同步序是对程序之间的冲突访问的定序。因此程序序与同步序结合起来就完成了对整个并行程序的访存事件的定序。

定义 8　设 $E_S(PRG)$ 是流并行程序 PRG 在流一致性模型 S 中的一个执行, 该执行的同步序与程序序的并集称为该执行的发生序, 记为 $HB_S(E_S(PRG))$, 即 $HB_S(E_S(PRG)) = PO(PRG) \cup SO_S(ES(PRG))$。

并非所有执行所确定的访存次序都是满足程序员要求的。执行引入了程序或线程之间的访存次序关系, 这种次序关系必须和程序内部固有的访存次序关系相一致, 否则就不能满足程序员的要求[7]。因此, 可以这样定义正确的执行:

定义 9　执行 $E_S(PRG)$ 是正确的当且仅当 $HB_S(E_S(PRG))$ 无圈。

Scheurich[9] 给出了判断一个并行执行是否正确的标准, 它规定一个正确的并行执行必须得到与串行执行相同的结果。不难证明, 定义 9 所定义的正确执行是满足这个标准的。

对于一个程序而言, 正确的存储一致性模型应该为程序中所有的冲突访问定序。反之, 判断一个程序是否符合存储一致性模型的编程规范的标准, 是该程序中所有冲突访问是否都能够被存储一致性模型定序[8]。可以由此给出判断一个流并行程序是否满足流一致性模型 S 的标准。

定义 10　流并行程序 PRG 符合流一致性模型 S 的要求, 当且仅当该程序在 S 中存在正确的执行

且在任一正确执行 $E_S(PRG)$ 下,程序中所有冲突的流访问操作都能被 $HB_S(E_S(PRG))$ 定序。

对于一个系统而言,如果正确实现了某个存储一致性模型,那么该模型下的正确程序在该系统下都能够得到正确的执行结果。前面已经提到,程序的执行结果由冲突访问执行的次序决定,同一程序内的冲突访问的执行次序由程序序决定,程序之间的冲突访问次序由同步操作的执行次序决定,而同步操作的执行次序又是由执行 $E_S(PRG)$ 来决定。单机的程序序一般是能够保持的。因此,一个正确实现了某个存储一致性模型的系统,必须保证错误的执行都不出现。由此给出一个系统是否满足流一致性模型 S 的标准。

定义 11 一个系统满足流一致性模型 S 的要求,当且仅当在该系统下,S 中正确程序的错误执行都不会发生。称该系统为流一致性模型的一个正确实现。

4 流一致性模型的正确实现及证明

在第 2 节提出的限制条件中,操作完成通常是指该操作所产生的效果对所有的处理机可见,这是一个充分条件而非必要条件。在很多情况下,并不需要将操作的结果对所有的处理机公开,只需要部分对操作结果感兴趣的处理机可见就行了。此外,限制条件并没有具体指明同步序和程序序在物理上先后发生的意义。这部分内容将由存储一致性模型的实现来给出,或者说,存储一致性模型的实现是对发生序的具体化。下面用 u^i 表示访存操作 u 相对于处理机 P_i 完成[4],$u^i < v^j$ 表示事件 u^i 在事件 v^j 之前发生。

首先考虑同步操作之间的顺序,限制条件规定"必须且仅需保持同一组(rel, acq)操作的顺序"。对于(rel, acq) $\in RA$,由于流编程模型中冲突访问的非独占性,只要发出 acq 的处理机看到 rel 完成,就可以发出 acq 操作,因此有

$$\mathrm{rel} \xrightarrow{E_S} \mathrm{acq} \Rightarrow \mathrm{rel}^c < \mathrm{acq}^c \tag{1}$$

其中,(rel, acq) $\in RA$,c 是发出 acq 操作的处理机。

(1)式的含义是,如果 rel 被 $E_S(PRG)$ 定序在 acq 之前完成,那么程序在该系统实现下的任意一次执行中,rel^c 都在 acq^c 之前发生,即 rel 相对于处理机 c 在 acq 之前完成。

对于读操作与同步操作之间的顺序,限制条件规定"在任何共享流的读取开始之前,与其相关联的 acq 操作必须完成"。只要 acq 操作完成,处理机就可以发出读操作 sl,与其他处理机无关,所以有:

$$\mathrm{acq} \xrightarrow{PO} \mathrm{sl} \Rightarrow \mathrm{acq}^c < \mathrm{sl}^c \tag{2}$$

其中,c 是发出 acq 操作和 sl 操作的处理机。

对于写操作与同步操作之间的顺序,限制条件规定"在任何 rel 操作执行之前,与其相关联的共享流的写回必须完成"。在发出 rel 操作之前,必须保证写操作 ss 相对所有共享此流的处理机都已完成,所以有:

$$\mathrm{ss} \xrightarrow{PO} \mathrm{rel} \Rightarrow \mathrm{ss}^i < \mathrm{rel}^c \tag{3}$$

其中,c 是发出 rel 操作的处理机,$i \in \mathrm{share(ss)}$,share(ss) 表示所有共享 ss 所操作的流的处理机集合。

综合(1)~(3)式,对于(ss, sl) $\in SO_S(E_S(PRG))$,有

$$\mathrm{ss} \xrightarrow{SO_S} \mathrm{sl} \Rightarrow \mathrm{ss}^i < \mathrm{sl}^c \tag{4}$$

其中,c 是发出 sl 操作的处理机,$i \in \mathrm{share(ss)}$。

(4)式的含义是,如果 *ss* 操作被同步序 $SO_S(E_S(PRG))$ 定序在 sl 操作之前完成,那么程序在该系统实现下的任意一次执行中,ss^i 在 sl^c 之前发生,即流的写回相对所有共享此流的处理机都完成之后,处理机 c 才能完成对此流的读操作。

最后,考虑 ss 操作和 sl 操作之间的程序序,同一个处理机内部的访存操作只要相对自己保持顺序即可,因此有

$$u \xrightarrow{PO} v \rightarrow u^c < v^c \tag{5}$$

其中, u、v 是 ss 或者 sl 操作, c 是发出 u、v 操作的处理机。

(4) 式和(5) 式合起来就是流一致性模型的一个实现对发生序的具体化:

$$\begin{cases} \mathrm{ss} \xrightarrow{SO_S} \mathrm{sl} \Rightarrow \mathrm{ss}^{i} < \mathrm{sl}^{c} \\ u \xrightarrow{PO} v \Rightarrow u^{d} < v^{d} \end{cases} \tag{6}$$

其中, c 是发出 sl 操作的处理机, $i \in \mathrm{share}(\mathrm{ss})$, u、v 是 ss 或者 sl 操作, d 是发出 u、v 操作的处理机。

定理 1　(6) 式所表示的访存次序限制条件是流一致性模型的一个正确实现。

证明　根据定义 11, 流一致性模型 S 正确实现的条件是, 该实现能够保证 S 中正确程序的错误执行都不会发生。设 $E_S(PRG)$ 是 S 中正确程序 PRG 的一次错误执行, 这次执行将导致发生序 $HB_S(E_S(PRG))$ 中有圈, 记为 δ。由于 $PO(PRG)$ 是无圈的, 因此 δ 中一定包含 $SO_S(E_S(PRG))$ 中的元素, 不妨设 $\mathrm{ss}_0 \xrightarrow{SO_S} \mathrm{sl}_0$ 且 $(\mathrm{ss}_0, \mathrm{sl}_0) \in \delta$。设 x 为 δ 中 sl_0 的下一操作, 即 $\mathrm{sl}_0 \xrightarrow{HB_S} x$。由于 (sl_0, x) 不可能属于 $SO_S(E_S(PRG))$, 因此只有 $\mathrm{sl}_0 \xrightarrow{PO} x$。再考察 x 在 δ 中的下一操作, 不难发现, 若要在 $HB_S(E_S(PRG))$ 中形成圈, δ 只可能是以下这种形式:

$$\mathrm{ss}_0 \xrightarrow{SO_S} \mathrm{sl}_0 \xrightarrow{PO} \cdots \xrightarrow{PO} \mathrm{ss}_1 \xrightarrow{SO_s} \mathrm{sl}_1 \xrightarrow{PO} \cdots$$

$$\cdots \xrightarrow{PO} \mathrm{ss}_j \xrightarrow{SO_S} \mathrm{sl}_j \xrightarrow{PO} \cdots \xrightarrow{PO} \mathrm{ss}_0$$

将(6) 式所确定的访存次序限制条件代入上式, 最后可得 $\mathrm{ss}_0^{i} < \mathrm{ss}_0^{d}$, $i \in \mathrm{share}(\mathrm{ss})$。显然 $d \in \mathrm{share}(\mathrm{ss})$, 因此有 $\mathrm{ss}_0^{d} < \mathrm{ss}_0^{d}$, 而这是不可能发生的。所以在实现了(6) 式的系统中, S 中正确程序 PRG 的错误执行 $E_S(PRG)$ 是不会发生的。得证。

5　结 论

在分析流编程模型下流程序的访存行为的基础上, 提出了流一致性模型, 它比传统的 RC 模型更加松弛, 因而可能减少流编程环境下的数据一致性维护开销。本文还讨论了流一致性模型下正确程序的设计标准和正确系统实现的标准, 给出了流一致性模型的一个正确实现, 并完成了正确性证明。下一步的工作将考虑更加复杂的数据共享情况和如何松弛一条数据流内部的数据访问, 以及如何对流一致性模型进行性能评测。

参 考 文 献:

[1] Lamport L. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs [J]. IEEE Transactions on Computers, 1979, 28: 241- 248.

[2] Goodman J R. Cache Consistency and Sequential Consistency [R]. Technical Report no. 61, SCI Committee, 1989.

[3] Dubois M, Scheurich C, Briggs F. Memory Access Buffering in Multiprocessors [C]// Proceedings of 13th Annual International Symposium on Computer Architecture, 1986: 434- 442.

[4] Gharachorloo K, Lenoski D, Laudon J, et al. Memory Consistency and Event Ordering in Scalable Shared-memory Multiprocessors[C]// Proceedings of 17th Annual International Symposium on Computer Architecture, 1990: 15- 26.

[5] Mattson P. A Programming System for the Imagine Media Processor [D]. Ph. D. Dissertation, Department of Electrical Engineering, Stanford University, 2002.

[6] Patterson D A, Hennessy J L. Computer Architecture: A Quantitative Approach (Second Edition) [M]. Beijing: China Machine Press, 1999.

[7] Shasha D, Snir M. Efficient and Correct Excution of Parallel Programs that Share Memory [J]. ACM Transactions on Programming Languages and Systems, 1988, 10: 282- 312.

[8] 胡伟武. 共享存储体系结构 [M]. 北京: 高等教育出版社, 2001.

[9] Scheurich C, Dubois M. Correct Memory Operation of Cache-based Multiprocessors [C]// Proceedings of 14th Annual International Symposium on Computer Architecture, 1987: 234- 243.