面向多核程序确定性重演的内存竞争记录机制研究

朱素霞[1],季振洲[1],刘 涛[1],王 庆[1],张 浩[2]
(1.哈尔滨工业大学计算机科学与技术学院,黑龙江哈尔滨 150001;
2.中国科学院计算技术研究所,北京 100190)

摘 要: 内存竞争记录是实现多核程序确定性重演的关键技术.针对现有内存竞争记录算法存在消耗资源多、记录日志大、重演速度受限等问题,本文提出了一种硬件结构支持的、基于分段技术的、高效的点对点内存竞争记录算法,该算法用一种更严格的间接发生序表示内存竞争,采用分段的可推导约减算法减少记录内存竞争的次数,同时使用最大近似时戳法近似已被替换出 cache 的内存块的时戳,能够在引入较少硬件资源的前提下记录较小的内存竞争日志,且简单易实现,重演速度快.

关键词: 多核程序; 确定性重演; 内存竞争记录
中图分类号: TP303 **文献标识码:** A **文章编号:** 0372-2112 (2011) 12-2748-07

Study on Memory Race Recording Mechanism in Deterministic Multi-Core Replay

ZHU Su-xia[1], JI Zhen-zhou[1], LIU Tao[1], WANG Qing[1], ZHANG Hao[2]
(1. *School of Computer, Harbin Institute of Technology, Harbin, Heilongjiang* 150001, *China*;
2. *Institute of Computing Technology Chinese Academy of Sciences, Beijing* 100190, *China*)

Abstract: Memory race recording is a key technology in deterministic multi-core replay. High hardware consumption, large log size and slow replay speed limited the application of previous memory race recording algorithms. This paper proposes a new efficient hardware memory race recording algorithm, which is implemented in chunks and logs the outcomes of memory races in a point-to-point approach. In this recording mechanism, memory race is presented in a new stricter indirect dependency, a chunk-based transitive reduction algorithm is introduced to reduce the number of memory races logged, a maximum timestamp approximation method is proposed to deal with those races evicted from cache. This new memory race recording algorithm can replay a program at production run speed, and log a smaller memory race log using smaller hardware than previous point-to-point approaches.

Key words: multi-core program; deterministic replay; memory race recording

1 引言

随着多核处理器的流行,多核程序的应用越来越广泛.然而,多核程序运行的结果存在不确定性,给程序调试、容错处理、入侵检测等应用带来了众多挑战,也制约了并行计算的发展[1].多核程序确定性重演技术通过记录多核程序运行时的不确定性信息,能够解决多核程序运行的不确定性,具有广泛的应用前景[2].为了实现多核程序的确定性重演,需要在程序原始执行时记录足够的不确定信息,这些信息主要包括:(1)程序的初始状态;(2)程序的输入;(3)线程间的内存竞争.记录前两种信息的方法与单核处理器相同,目前已有众多软件[3~5]实现了对它们的记录,而且在现有计算机系统中,带来较小的性能开销.然而,现代多核计算机大都支持线程间快速的通讯机制,内存竞争频发,使得内存竞争记录成为实现多核程序确定性重演的关键技术.基于软件实现的内存竞争记录器[6,7]存在记录的信息量大、带来的开销大、对原有系统性能影响大等问题.而硬件实现的内存竞争记录器与软件实现相比,具有低成本、低开销、高性能等优点,因此,有些研究者提出采用相对廉价的硬件结构实现内存竞争的记录[8~13],从而在体系结构级支持多核程序的确定性重演.

收稿日期:2010-12-15;修回日期:2011-05-03
基金项目:国家自然科学基金(No.60736012)

2 相关工作

基于硬件的点到点的内存竞争记录算法需要跟踪每一个内存操作,记录由冲突双方对应的指令计数值表示的内存竞争发生序到内存竞争日志中.如 FDR[8]记录内存冲突 $i:x \rightarrow j:y$ 对应的内存竞争发生序 $x \rightarrow y$ 到内存竞争日志中,其中,i、j 表示线程的 ID 号,x、y 表示冲突双方指令的动态指令计数值(IC,dynamic Instruction Count),$\rightarrow$ 表示冲突双方发生的先后顺序.要实现多核程序的确定性重演,需要记录下足够的内存竞争发生序,但并不需要记录所有内存竞争发生序.FDR 中使用 TR 方法[7]只记录那些不能通过先前已记录的内存竞争发生序推导出来的内存冲突对应的内存竞争发生序,减小了记录内存竞争的次数.RTR[10]通过使内存竞争发生序规则化、严格化,创建了更为紧凑的内存竞争日志.这种点到点的记录方式可以实现多核程序的快速重演,具有广泛的应用前景.但是,为了记录内存竞争,FDR 和 RTR 均为每个 cache 块添加一个用最近访问该内存块指令的 IC 表示的时戳,带来了较多的硬件资源消耗:假设 IC 用 64bits 表示,对于包含 1K 个 cache 块的一级数据 cache,则至少需要为添加 8KB 硬件资源来存储时戳.

基于硬件的 chunk 记录方式不再记录内存竞争发生序,而是记录由单个线程内没有与其他线程发生冲突的连续执行的多条指令组成的 chunk 及其时戳.这种记录方式通常使用硬件签名寄存器实现,增加的硬件资源较少.Rerun[12]和文献[13]都采用这种方式记录内存竞争日志.虽然 Rerun 记录的日志大小可以同 RTR 相媲美,但在实现重演时,只能按照 chunk 时戳顺序地重放,大大减缓了重演的速度,难以应用到对重演速度要求较高的场合,如容错处理[2].文献[13]采用了并发 chunk 域,在一定程度上提高了重演的速度,但却增大了内存竞争日志.同时,硬件签名寄存器也存在一定的设计复杂度.

针对上述内存竞争记录方式中存在的问题,本文提出一种硬件结构支持的、基于分段技术的、高效的点到点内存竞争记录算法(CPMR—Chunk-based Point-to-point Memory race Recording Algorithm),该算法能够在降低硬件资源消耗的前提下,记录更小的内存竞争日志,且简单、易实现、重演速度快.

3 基于分段的点到点内存竞争记录算法

3.1 用间接发生序表示内存竞争

本文提出的基于分段的点到点内存竞争记录算法采用一种更严格的间接发生序——冲突发生时冲突双方所在处理器核的当前指令间的依赖关系,来表示内存竞争.间接发生序用依赖关系 $w \rightarrow v$ 来表示,w、v 表示发生冲突 $i:x \rightarrow j:y$ 发生时,线程 i、j 的当前指令计数值(CIC,Current dynamic Instruction Count).如图 1(a)所示,有两个线程 i 和 j 都对 z 执行写操作,线程 j 先对 z 进行写操作(IC = 1),之后线程 i 再对 z 执行写操作(IC = 3),这时检测到内存冲突 $j:1 \rightarrow i:3$,而此时线程 j 已执行完 IC = 2 的指令,即 CIC = 2,在记录内存竞争时,不再记录准确的内存竞争发生序 1 → 3,而是记录 2 → 3 这个间接发生序.同样,线程 i、j 间的其他内存竞争同样可以采用这种间接发生序来表示,如图 1(a)中 Indirect Log 给出了采用间接发生序表示内存竞争时所记录的内存竞争日志,共包含 6 个依赖关系.

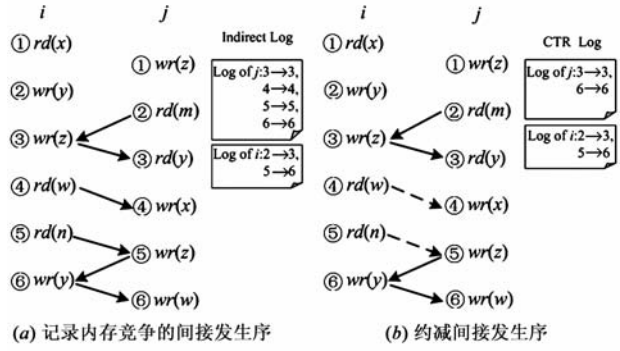


图1 用间接发生序表示的内存竞争

采用这种间接发生序表示内存竞争不需要知道已执行过的内存操作指令对应的 IC 值,也就不需要再为每个内存块保存对应内存操作的 IC 值.同时,这种间接发生序不止能够充分的指出线程间冲突发生的先后顺序,而且它更严格的指出了程序原始执行过程中冲突发生时线程间指令运行的先后顺序,是冲突发生的充分条件.因为:$i:w$ 总是会在 $i:x$ 之后发生或等于 $i:x$,而 $j:v$ 总是等于 $j:y$.因此,通过记录线程间内存冲突的间接发生序,完全可以使程序能够按原始的执行路径重放,实现确定性重演.

3.2 分段的可推导约减算法

基于分段的点到点内存竞争记录算法采用间接发生序表示内存竞争,同样存在某些内存冲突可以通过已记录的间接发生序推导出来.如图 1(b)所示,内存冲突 $i:1 \rightarrow j:4$ 可以由间接发生序 3 → 3 推导出来,内存冲突 $i:3 \rightarrow j:5$ 也可以由间接发生序 3 → 3 推导出来,依此类推,只要记录下那些不可推导的间接发生序,如图 1(b)中 CTR Log 所示,就可以实现程序的确定性重演.而且,因为间接发生序是冲突发生序的充分条件,可以使得间接发生序推导出更多的内存冲突,从而可以在更大程度上减小内存竞争日志.

为了使该内存竞争记录算法能快速的识别并记录那些不可推导的内存冲突的间接发生序,本文提出了

分段的可推导约减算法(CTR—Chunk-based Transitive Reduction).从图 1(b)中可以看出,只要记录一个间接发生序后,下次再检测到冲突时,如果冲突的先发生方发生在已记录下的间接发生序的先发生方之后,则此冲突不能由先前记录的间接发生序推导出来,需要将其对应的间接发生序记录到内存竞争日志中;否则约减掉.为了只记录那些不能够推导出来的内存冲突的间接发生序,需要在记录间接发生序时设置一个标志,以用来判断下次检测到的冲突的先发生方是否在此标志之后发生.为此,本文引入分段技术,将线程 i、j 内的指令进行分段,除最开始的一个段外,其他段的开始和结束都源于间接发生序的记录.并且每个处理器核都增加一个段计数器来记录段的时戳,以标记段创建的先后顺序.这样,每个线程被划分为多个段,每个段都由多条指令组成,每条指令都有其对应的段时戳.通过判断冲突的先发生方所在段的时戳是否不小于最新段时戳,来决定此冲突的间接发生序是否需要记录.如图 2(a)所示,当线程 i 开始运行时,一个新段 0 创建;当检测到冲突 i:2 → j:3 后,需要记录下它对应的间接发生序 3 → 3,此时,旧段 0 结束,新段 1 创建,i:3 及 i:3 以前的指令都标记为段 0,而之后的指令标记为段 1;当检测到冲突 i:1 → j:4 时,因为 i:1 位于段 0 内,小于线程 i 标记的最新段 1,即此冲突可以推导出来,所以此冲突的间接发生序就可以约减掉,同样,冲突 i:3 → j:5 对应的间接发生序也可以约减掉;当检测到冲突 i:4 → j:6 时,因为 i:4 在新段 1 内,从而需要记录它对应的间接发生序 6 → 6,此时,旧段 1 结束,新段 2 创建;依次类推,就可以记录下线程 i、j 间足够的间接发生序,供程序实现确定性重演.

上面描述了系统中只有两个线程的情况,如果系统中有更多线程,并且每个线程都运行在不同的处理器核上,则需要每个处理器核分别为其他的每个处理器核记录一个最新段时戳.假设系统中共有 P 个处理器核,每个处理器核中不只增加一个段计数器来标记本处理器核对应线程内的段时戳,还要分别为其他 $P-1$ 个线程保存的一个最新段时戳,以便在检测到冲突时用来判断是否需要记录冲突的间接发生序.一旦 j 线程中需要记录它同 i 线程之间一个间接发生序,则 i 线程所记录的段时戳计数器加 1,i 中所记录的 j 线程的最新段时戳也要更新为 i 线程的当前段时戳,而 i 线程中所记录的其他线程的最新段时戳保持不变.依此类推,就可以在多核程序运行的过程中记录下足够的内存竞争间接发生序,供程序实现确定性重演.多线程间的记录示例如图 2(b)所示.

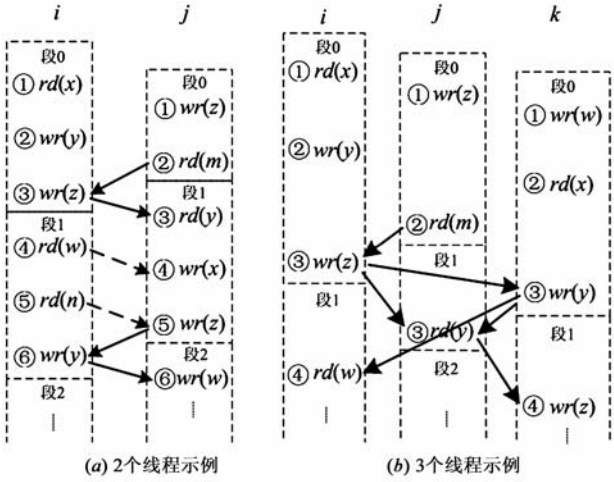


图2 采用CTR约减内存竞争日志

3.3 基于分段的点到点内存竞争记录算法对重演的影响

当多核程序按照记录的日志重演时,需要每个线程按照已记录的内存竞争日志进行检查,以便能够按照程序原始执行时的路径重放.基于分段的点到点记录算法为每个内核记录一个点到点的内存竞争日志,可以使得重演时各线程并行地重放,实现快速的重演.只有在 i:w 还没有完成,而 j:v 就已经准备好这种特殊情况下,这些检查才会使得操作 j:v 的执行放缓.然而,该算法在更大程度上减小了记录内存竞争的次数(见第 7 节的仿真结果),从而能在更大程度上减少重演时需要进行检查的次数,因此,在另一方面,该算法又会增强重演的性能.具体对重演性能的量化评估,会在将来的工作中进行.

4 内存竞争的检测与记录

基于分段的点到点内存竞争记录算法同 FDR 和 RTR 一样,通过简单的修改原有的 cache 一致性协议来检测并记录内存竞争.但是,该算法中的冲突检测在应答方进行,间接发生序的记录在请求方进行.因此,只有在检测到冲突时,才会在应答消息中添加一个冲突记录标志位和当前指令计数值 CIC.冲突检测与记录的过程描述如下:请求方因为有共享内存操作向一致性协议机构发出请求消息;一致性协议机构收到请求消息后,将此请求消息转发给其他处理器核,即应答方;应答方采用分段的可推导约减算法进行冲突检测,当检测到需要记录的冲突时,将一个冲突记录标志位和当前指令计数值 CIC 连同内存块的内容一起发送给请求方,若没有检测到冲突,则只将内存块的内容发送给请求方;请求方接收到应答消息后,首先检测冲突记录标志位是否为真,如果为真,则记录下此冲突对应的间接发生序,否则不记录.

该算法基于目录一致性协议的内存竞争记录的具体过程如图 3 所示,带黑圈的数字标出了操作步骤.两

个线程 i 和 j 分别运行在处理器核 P1 和 P2 上,线程 i 首先对 x 进行写操作,之后线程 j 又对 x 进行写操作.当 P1 写 x 后,x 所在 cache 块的状态为 M 状态,当 P2 写 x 时,向一致性协议机构发出一致性请求 GETX,目录收到此消息后,再将此请求消息转发给 P1,P1 收到请求消息后,结合消息的类型和自身存储的变量 x 的状态,检测到冲突(写之后又被写),此时 P1 更新线程 i 的段时戳及它所记录的 j 线程的最新段时戳,并将冲突记录标志和 P1 当前的指令计数值 CIC(此时 CIC = 3)连同 x 的内容一同发送给请求方 P2;P2 收到应答消息后,首先检测冲突记录标志是否为真,如果为真,则记录此冲突的间接发生序 3 → 2 到 j 线程的内存竞争日志中.

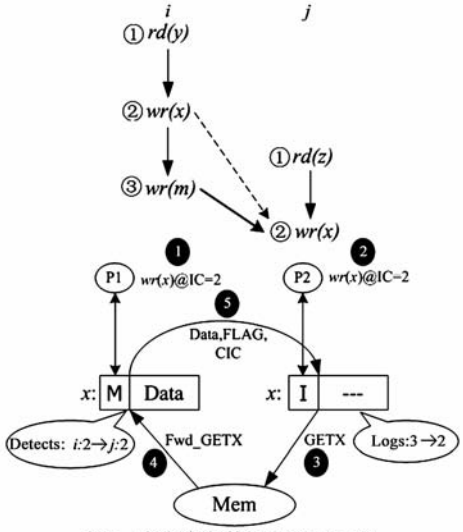


图3　内存竞争检测与记录示例

5　记录已被替换出 Cache 的内存竞争

现实的处理器中,cache 的容量总是有限的,只能够存储部分内存块,致使丢失对先发生方已被替换出 cache 的内存竞争的记录.在 FDR 中总是用当前指令计数值 CIC 来近似已经被替换出 cache 的内存块的时戳,从而降低了硬件开销,而 CIC 总是大于或等于内存块的真实时戳,因此,对应的内存竞争的发生序不可能被约减掉.RTR 采用 set/LRU 近似算法,总是使用本组 cache 中最小的时戳来近似已经被替换出 cache 的内存块的时戳,这样虽然可以约减掉一部分发生序,但每次都需要访问 cache 来寻找当前最小时戳,增加了 cache 开销.因此,本文提出一种新的近似方法—最大近似时戳法:用历史被替换出 cache 的内存块的最大段时戳,来近似已被替换出 cache 的内存块的段时戳.这个近似的段时戳一定是某个先前被访问过的内存块对应的段时戳,它也一定小于或等于当前指令对应的段时戳,也一定大于或等于已被替换出 cache 的内存块的准确段时戳.如果它也小于对应处理器核所记录的最新段时戳,则此内存竞争对应的间接发生序就可以约减掉,从而记录更少的间接发生序,减小内存竞争日志.所以,只要发生替换操作,都要进行比较并记录下历史最大段时戳;只要每次检测到内存竞争,都要判断这个近似段时戳是否大于记录的最新段时戳,如果大于,则记录此内存竞争的间接发生序到内存竞争日志,否则约减掉.因此,本方法不但减少了硬件资源消耗,降低了 cache 的开销,还可以采用分段的可推导约减算法来减少记录内存竞争的次数.

举例说明,如图 4 所示,当 i 线程写 z 时,因为 cache 满,替换出 y,当 j 线程读 z 时,检测到冲突,需要记录下间接发生序 3 → 3,当 i 线程读 w 时,替换出 x,当 j 线程写 x 时,又检测到冲突,但此时 x 已被替换出 cache.如果用已替换出 cache 的 x 和 y 中段时戳的最大值来近似 x 的段时戳,即近似时戳为 1,而当前线程 i 的段时戳值已记录到 3,因此,不需要记录此冲突的间接发生序,从而减小了内存竞争日志.

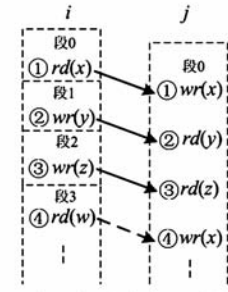


图4　最大近似时戳法约减内存竞争

6　基于硬件的算法描述及具体实现结构

本文提出的采用分段技术实现的点对点内存竞争记录算法基于硬件的描述如表 1 所示,它详细描述了每个处理器核的状态和动作.

该算法中每个处理器做如下动作:

(1)指令提交时,如果是内存操作(store 或者 load 指令),更新 IC 值,并设置对应内存块的段时戳.

(2)应答方接收到一致性请求后,先判断是否有冲突发生;如果检测到冲突,通过基于分段的可约减算法判断此冲突的间接发生序是否需要记录;如果需要记录,则在发送给请求方的应答消息中添加冲突记录标志位和 CIC 值,并结束旧段,创建新段,更新对应请求方的最新段时戳.

(3)请求方收到一致性应答消息后,先判断冲突记录标志位是否为真;如果为真,则记录间接发生序到该请求方的内存竞争日志.

该算法具有如下突出特点:(1)用段时戳表示每个内存块的时戳;(2)采用分段的可推导约减算法减少记录的次数;(3)在应答方进行内存竞争检测,在请求方进行内存竞争记录;(4)仅在需要记录内存竞争时,向应答消息中添加冲突记录标志位和 CIC 值;(5)用最大近似时戳法近似已被替换出 cache 的内存块的时戳;(6)该算法引入的硬件结构比较少,且简单易实现.

表 1 CPMR 基于硬件的内存竞争算法描述

处理器核 i 增加的状态：
IC：动态指令计数；
LOG：与其他每个处理器的内存竞争日志缓存入口；
CCTS[M]：一级数据 cache 中每个 cache 块的段时戳，M 为 cache 中 cache 块的数目；
GCTS：全局段时戳；
VCTS[$P-1$]：为其他每个处理器核所记录的最新段时戳，P 为处理器的总数；
GPCTS：最大近似时戳.

处理器核 i 的动作：

```
/* 有内存操作时 */
On commit of load/store instr of block b {
    IC ++;
    CCTS[b] = GCTS;
}
/* 内存块被替换出 cache 时 */
On replacing bock b {
    if(CCTS[b] > GPCTS){
        GPCTS = CCTS[b];
    }
}
/* 收到其他处理器核 k 发送过来的一致性应答消息时 */
On receiving a coherence reply for block b from proc k {
    (ID,FLG,CIC): = RECRIVE( );
    if (FLAG = = conflict_bit)
        APPEND(LOG,ID,IC,CIC);
}
/* 收到其他处理器核 j 发送过来的一致性请求消息时 */
On receiving a coherence request for block b from proc j {
    if(b isn't present in cache) {
        if (GPCTS > VCTS[j]) {
            GCTS ++;
            VCTS[j] = GCTS;
            SEND(ID,FLAG,CIC,…);
        }
    }
    else{
        if ((CONFLICTS( ))){
            if (CCTS[b] > = VCTS[j]) {
                GCTS ++;
                VCTS[j] = GCTS;
                SEND(ID,FLAG,CIC,…);
            }
        }
    }
}
```

结合上述硬件描述算法，图 5 给出了基于 4 核 CMP 的硬件实现结构框图.基于该算法的内存竞争记录器为每个处理器核添加如下部件：一个动态指令计数器 IC(64bits)；一个内存竞争日志缓冲区入口地址寄存器 LOG(32bits)；一个全局段时戳寄存器 GCTS(56bits)，用来存储本处理器核记录的最大段时戳；一个最大近似时戳寄存器 GPCTS(56bits)，用来存储被替换出 cache 的内存块的历史最大段时戳；一个段时戳向量 VCTS，其中的每个元素(56bits)表示为其他每个处理器核所记录的最新段时戳.为每个处理器核的 L1 DCache 中的 cache 块增加一个段时戳 CCTS(56bits)，用来存储其对应的段时戳.

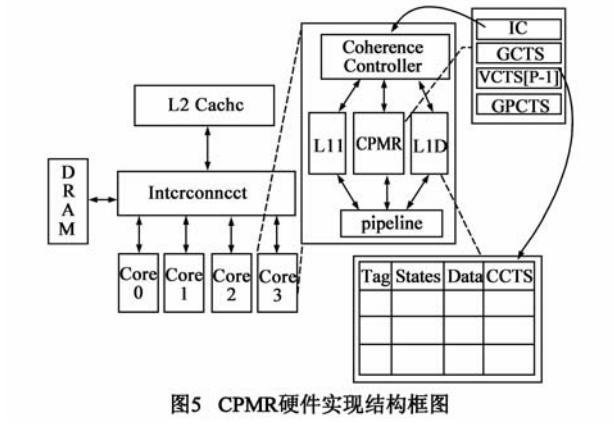


图5 CPMR硬件实现结构框图

7 仿真结果

本文使用 Wisconsin GEMS 全系统仿真器[14]来仿真该内存竞争记录算法.GEMS 通过使用 Simics 模拟一个企业版的 SPARC 多核处理器，同时可以运行 Solaris10 操作系统.表 2 给出了仿真器的关键配置参数.本文选取 SPLASH2[15]并行测试负载进行测试.SPLASH2 是典型的多线程科学计算应用，需要频繁地访问共享变量.

表 2 GEMS 关键配置参数

参数	说明
Cores	4 核，in-order，3GHZ
Private L1 Cache	分离的指令 cache 和数据 cache，大小为 64KB、4 路组相关、采用写回策略、每个 cache 块为 64B，LRU 替换策略
Shared L2 Cache	8M 8 路组相关，LRU 替换策略
Memory	4G DRAM
Coherence	MESI 目录一致性协议，silent 替换策略
Consistency Model	Sequential Consistency(SC)

下面给出 CPMR 在硬件资源消耗、日志记录及性能开销方面的仿真结果.

7.1 硬件资源消耗

图 6 是对测试负载中段数目统计结果，可以看出平均每个段约包含 2000 条内存操作指令，段的计数值远小于 IC 的计数值.所以，如果时戳是以字节为单位，且 IC 用 64bits 来表示，则完全可以用 56bits 来表示段时戳.因此，可以减少所需要增加的 cache 及寄存器的容量.具体硬件资源的消耗如表 3 所示，CPMR 仅需要为 cache 增加 7KB 容量，相比 RTR 减少了 70% 的硬件消耗.

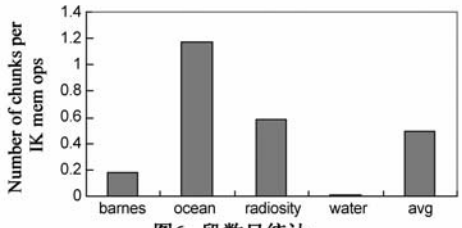


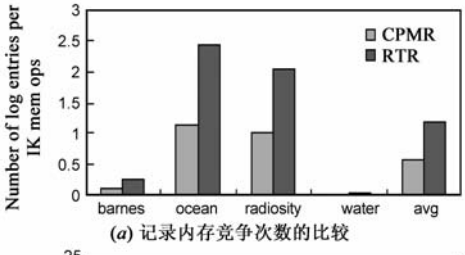
图6　段数目统计

表 3　CPMR 与 RTR 硬件资源消耗比较

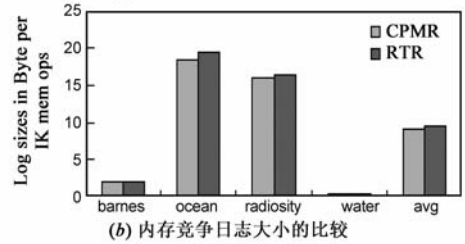
记录器	每个处理器核增加的硬件	Cache 增加量	寄存器增加量
RTR	时戳存储单元、动态指令计数器、指令计数向量、滑动窗口	24KB	80B
CPMR	段时戳存储单元、动态指令计数值、段时戳计数器、段时戳向量、历史最大段时戳寄存器	7KB	43B

7.2　日志记录

图 7 给出了 CPMR 同 RTR 在内存竞争日志方面的比较.从图 7(*a*)可看出,CPMR 记录内存竞争的次数是 RTR 的 48%,原因是 CPMR 用间接发生序表示内存竞争,采用分段的可推导约减算法,能更大程度上减少记录内存竞争的次数.从图 7(*b*)可看出,CPMR 平均每 1K 个内存操作记录约 9Bytes 的内存竞争日志,比 RTR 减少了约 4%;从文献[13]中给出的 Rerun 与 RTR 比较结果,还可间接的得出 CPMR 记录的日志比 Rerun 更小.



(*a*) 记录内存竞争次数的比较

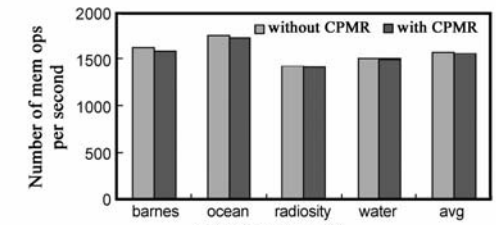


(*b*) 内存竞争日志大小的比较

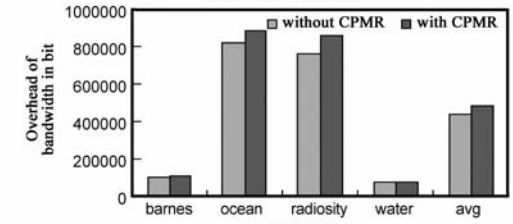
图7　内存竞争日志比较

7.3　性能开销

图8(*a*)给出了 CPMR 的运行时间开销分析,带 CPMR 的 4 核 CMP 在性能上降低了约 1%,而 RTR 和 Rerun 的开销都在 2%,这是因为 CPMR 记录日志的次数大大减少,也相应的降低了因记录内存竞争而带来的开销.CPMR 同 RTR、Rerun 一样,都向一致性协议消息中添加了附加消息字段,都增加了原有系统的带宽,但 CPMR 由于只在需要记录内存竞争时才向一致性协议消息中添加负载,从而带来了较小的带宽开销,如图8(*b*)所示,CPMR 平均带来了不到 9%开销,而 RTR、Rerun 的开销都在 10%.



(*a*) 运行时间开销



(*b*) 带宽开销

图8　性能统计

8　结论

本文针对现有内存竞争记录中存在的问题,提出了一种硬件支持的、基于分段的、高效的点对点内存竞争记录算法.该算法采用一种更严格的间接发生序表示内存竞争,为每个内存操作存储其对应的段时戳,减少了硬件资源开销;并使用基于分段的可推导约减算法,在更大程度上减少了记录内存竞争的次数;同时采用最大近似时戳法近似已被替换出 cache 的内存块的时戳,在增加较少硬件资源的情况下,记录较少的内存竞争.针对该算法在 GEMS 多核仿真平台下模拟实现了基于 4 核 CMP 的硬件内存竞争记录器,仿真结果表明:硬件资源消耗较低(~7KB/core),记录的内存竞争日志小(~9Bytes/1K mem ops),带来的时间开销小(~1%),带宽消耗低(<9%).

本文提出的内存竞争记录算法与已有的点到点内存竞争算法相比,具有如下几点优势:(1)记录的内存竞争次数少;(2)记录内存竞争日志小;(3)硬件资源开销低;(4)性能开销低;(5)更具有应用价值.

参考文献

[1] 杨际祥,谭国真,王荣生.多核软件的几个关键问题及其研究进展[J].电子学报,2010,38(9):2140-2146.
Yang Jixiang, Tan Guozhen, Wang Rongsheng. Some key issues and their research advances for multi-core software[J]. Acta Electronica Sinica, 2010, 38(9): 2140-2146. (in Chinese)

[2] C M Pancake, R Netzer. A bibliography of parallel debuggers, 1993 edition[A]. Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging (PADD)[C]. New York, USA: ACM, 1993. 169-186.

[3] T J Leblanc, J M Mellor-Crummey. Debugging parallel programs with instant replay[J]. IEEE Transactions on Computers, 1987, C-36(4): 471 – 482.

[4] L Levrouw, K Audenaert. Minimizing the log size for execution replay of shared-memory programs[A]. Third Joint International Conference on Vector and Parallel Processing[C]. Linz, Austria: Springer-Verlag, 1994. 76 – 87.

[5] D Lucchetti, S K Reinhardt, P M Chen. ExtraVirt: detecting and recovering from transient processor faults[A]. 2005 Symp on Operating System Principles Work-in-Progress Session[C]. Brighton, United Kingdom: ACM, 2005. 1 – 8.

[6] S Srinivasan, S Kandula, C Andrews, Y Zhou. Flashback: a lightweight extension for rollback and deterministic replay for software debugging[A]. Proceedings of the USENIX Annual Technical Conference[C]. Boston, Madison, USA: USENIX, 2004. 29 – 44.

[7] R H B Netzer. Optimal tracing and replay for debugging shared-memory parallel programs[A]. Proc of the ACM/ONR Workshop on Parallel and Distributed Debugging (PADD)[C]. San Diego, California, USA: ACM, 1993. 1 – 11.

[8] M Xu, R Bodik, M D Hill. A flight data recorder for enabling full-system multiprocessor deterministic replay[A]. Proc of the 30th Annual International Symposium on Computer Architecture[C]. San Diego, CA: ACM, 2003. 122 – 133.

[9] M Prvulovic. CORD: Cost-effective (and nearly overhead-free) order recording and data race detection[A]. Proc of the 12th IEEE Symp on High-Performance Computer Architecture[C]. New York, USA: IEEE Computer Society, 2006. 232 – 243.

[10] M Xu, R Bodik, M D Hill. A regulated transitive reduction (RTR) for longer memory race recording[A]. Proc of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems[C]. San Jose, California, USA: ACM, 2006. 49 – 60.

[11] S Narayanasamy, C Pereira, B Calder. Recording shared memory dependencies using strata[A]. Proc of the 12th International Conference on Architectural Support for Programming Languages and Op-erating Systems[C]. San Jose, California, USA: ACM, 2006. 229 – 240.

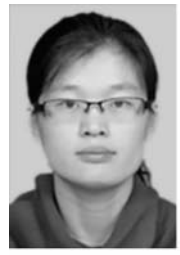
[12] D Hower, M Hill. Rerun: exploiting episodes for lightweight memory race recording[A]. Proceedings of the International Symposium on Computer Architecture[C]. Beijing, China: IEEE Computer Society, 2008. 265 – 276.

[13] G Pokam, C Pereira, K Danne, R Kassa, A Adl-Tabatabai. Architecting a chunk-based memory race recorder in modern CMPs[A]. Proceedings of the 42nd International Symposium on Microarchitecture[C]. New York, USA: ACM, 2009. 576 – 585.

[14] M M K Martin, D J Sorin, B M Beckmann, et al. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset[J]. Computer Architecture News, 2005, 33(4): 92 – 99.

[15] S C Woo, M Ohara, E Torrie, J P Singh, A Gupta. The splash-2 programs: characterization and methodological considerations[A]. 22nd Annual International Symposium on Computer Architecture[C]. Santa Margherita Ligure, Italy: ACM, 1995. 24 – 36.

作者简介



朱素霞 女,1978 年 3 月生于山东寿光,现为哈尔滨工业大学计算机科学与技术学院博士研究生,主要研究方向为计算机系统结构和并行计算.

E-mail: zhusuxia@hit.edu.cn

季振洲 男,1965 年 6 月生于黑龙江省哈尔滨市,现为哈尔滨工业大学计算机科学与技术学院教授、博士生导师,主要研究方向为计算机系统结构、并行计算和网络安全.

E-mail: jizhenzhou@hit.edu.cn