基于自适应释放策略的低开销确定性重放方法

应 欢[1],刘松华[2],唐博文[3,4],韩丽芳[1],周 亮[1]

(1.中国电力科学研究院有限公司,北京 100192;2.国网山东省电力公司东明县供电公司,山东 菏泽 274500;
3.中国科学院 计算技术研究所,北京 100190;4.中国科学院大学,北京 100049)

摘 要:针对基于页保护机制的确定性重放方法虽然能够有效降低记录开销,但由于页保护异常仍会引入性能开销的问题,本文深入研究了共享页面访问权限释放同步点对并行程序记录性能的影响,提出了一种基于自适应释放策略的确定性重放方法。采用PARSEC测试集进行性能评估,实验结果表明,该方法能够更进一步降低记录开销。

关键词:信息处理技术;并行程序;共享内存;调试;确定性重放;页保护

中图分类号:TP368 **文献标志码**:A **文章编号**:1671-5497(2018)06-1917-08

DOI:10.13229/j.cnki.jdxbgxb20171161

Efficient deterministic replay technique based on adaptive release strategy

YING Huan[1],LIU Song-hua[2],TANG Bo-wen[3,4],HAN Li-fang[1],ZHOU Liang[1]

(1.*China Electric Power Research Institute*,*Beijing* 100192,*China*;2.*State Grid Electric Power Company of ShanDong Province DongMing County Power Supply Company*,*Heze* 274500,*China*;3.*Institute of Computing Technology*,*Chinese Academy of Sciences*,*Beijing* 100190,*China*;4.*University of Chinese Academy of Sciences*,*Beijing* 100049,*China*)

Abstract: Inherent non-determinism in parallel program brings huge difficulty and challenge to programmers to test and debug. The major source of non-determinism is the undetermined order of shared memory accesses. Deterministic replay provides the possibility of reproducing parallel program execution, and deterministic replay method based on page protection mechanism can reduce the recording overhead effectively. However, the exception of page protection is the main performance cost. This paper deeply analyzes runtime overhead incurred by release synchronization points in record-runs, and proposes a deterministic replay method based on adaptive release strategy. Performance evaluation shows that the record slowdown is much smaller than prior user-level replay system.

Key words: information processing technology; parallel program; shared memory; debugging; deterministic replay; page protection

收稿日期:2017-11-29.

基金项目:"863"国家高技术研究发展计划项目(2015AA050202).

作者简介:应欢(1988-),女,工程师,博士.研究方向:程序分析与代码安全,动态优化.

E-mail:yinghuan1022@126.com

0 引 言

与传统的串行程序不同,并行程序具有不确定性,即在输入相同的情况下,同一个程序执行多次的结果并不相同。其中,造成并行程序不确定性的主要因素是各线程访问共享内存的顺序不确定[1]。为了提高并行程序的生产率,研究人员提出确定性重放技术。确定性重放技术包括记录和重放两步。在记录阶段,将各线程间的访存依赖关系记录在日志文件中;在重放阶段,强制各线程按照所记录的日志执行访存操作。部分研究工作[2-5]采用纯软件方法记录并行程序中共享内存访问的信息。这类方法保守认为只有将所有访问共享内存的信息全部记录下来,才能完整重现并行程序的记录执行。某些研究工作[6-10]探索在处理器中添加特殊的硬件来记录共享内存的访问顺序,由于有特殊硬件的支持,这些方法在记录阶段引入的运行开销几乎可以忽略。然而,这类方法需要修改当前成熟的处理器架构,相关的实验也是通过模拟器完成。鉴于设计复杂度和市场普及方面的考虑,目前硬件实现的确定性重放技术尚未得到商用处理器的支持。

为了降低时间开销,有研究人员[11,12]提出基于页保护机制的确定性重放方法。然而,Scribe需要修改操作系统内核,这可能会面临系统安全性、可移植性、可维护性等问题[13]。UPLAY[12]是中国科学院计算技术研究所体系结构国家重点实验室实现的低开销的确定性重放系统。该系统以页为粒度,通过记录并行程序中所有共享页面的访问权限在各线程间的转移顺序,来反映并行程序对共享内存的访问顺序。为了让持有页面访问权限的线程在确定的时机主动释放共享页面并避免线程间的死锁,UPLAY设定任何一个能被截获的执行点,都作为共享页面访问权限的释放同步点。然而,对于这种确定性重放方法来说,页保护异常是其主要的性能开销来源[14-16],预设多个共享页面访问权限的释放同步点,也可能会为并行程序引入处理页保护异常的开销,降低记录性能。

对此,本文首先深入分析了共享页面访问权限释放同步点对并行程序记录性能的影响;其次,提出一种基于自适应释放策略的确定性重放方法,基于该方法,实现了一个原型系统APLAY。采用PARSEC测试集[17]进行性能评估,结果表明,相比其他用户态下的确定性重放技术,本文方法能更进一步降低记录开销。

1 共享页面释放控制性能

在页保护机制的确定性重放方法中,页保护异常是主要的性能开销来源[14-16]。并行程序执行过程中触发页保护异常存在两种情况:①第一次访问某个共享页面;②释放后再次访问同一个共享页面。无法避免第①种情况下引入的页保护异常(每个新创建的线程执行前,所有共享页面都被添加页保护,否则将会遗漏某些访存)。与前者不同,通过优化释放策略,可以降低第②种情况下引入的页保护异常。本文对释放同步点的设置进行研究,并提出一种自适应的共享页面访问权限的释放策略,进一步降低并行程序处理页保护异常的开销。

目前,UPLAY[12]提供以下几种释放同步点的方法:①线程调用libc类库函数,例如文件操作、数学等相关的库函数;②线程调用Pthreads库函数,例如同步操作等相关库函数;③页保护异常处理,例如SIGSEGV信号处理函数入口、线程阻塞等待某个共享页面的访问权限等。本文以PARSEC测试集为例,分别统计并行程序在记录阶段、在原有释放同步点和减少部分释放同步点的情况下,各个测试程序处理页保护异常的时间开销,以及处理不同原因引入页保护异常所占的比例。

具体统计信息有:测试程序的整体运行时间t_{record}、处理页保护异常的时间t_{segv}、处理不同原因引入的页保护异常的时间(包括第一次访问共享页面触发页保护异常(t_{init})、libc类库函数处按需释放引入的页保护异常(t_{libc})、Pthreads类库函数按需释放引入的页保护异常(t_{Pthreads})、SIGSEGV信号处理函数入口按需释放引入的页保护异常(t_{handler})、线程阻塞等待某个共享页面的访问权限时按需释放引入的页保护异常(t_{request})、线程执行阻塞型库函数时被其他线程主动抢占引入的页保护异常($t_{\text{preempted}}$))。其中,测试情况1是指目前预设上述释放同步点进行性能测试统计,测试情况2是在情况1基础上减少libc类函数、Pthreads类库函数、SIGSEGV信号处理函数入口这3类释放同步点进行测试,因此情况2只有t_{init}、t_{request}、$t_{\text{preempted}}$这3种处理页保护异常开销的性能数据,具体统计结果如表1所示。

表 1　释放同步点对程序的性能影响

Table 1　Performance analysis for releasing synchronization points　　%

benchmarks	$(t_{record\_2}-t_{record\_1})/t_{record\_1}$	t_{segv}/t_{record}		t_{init}/t_{segv}		t_{libc}/t_{segv}		$t_{request}/t_{segv}$		$t_{pthreads}/t_{seg}$		$t_{handler}/t_{segv}$		$t_{preempted}/t_{segv}$	
		1	2	1	2	1	2	1	2	1	2	1	2	1	2
ferret	**17.4**	2.0	21.6	35.7	1.9	2.5	0.0	0.0	4.3	15.5	0.0	46.3	0.0	0.0	93.9
facesim	**−28.5**	24.1	10.3	4.8	9.8	20.8	0.0	5.3	43.2	0.4	0.0	6.6	0.0	7.5	34.1
raytrace	**−22.6**	39.2	23.5	5.3	16.6	0.0	0.0	40.0	74.1	0.0	0.0	54.7	0.0	0.0	0.0
bodytrack	**−5.3**	−4.1	9.0	79.0	70.5	0.5	0.0	2.2	6.9	1.5	0.0	8.9	0.0	1.3	0.7
vips	**3.6**	5.5	4.9	63.1	58.0	0.0	0.0	0.4	12.2	0.2	0.0	16.2	0.0	20.1	29.8
x264	**4.4**	0.0	0.0	65.5	10.0	0.0	0.0	0.3	0.0	31.8	0.0	2.5	0.0	0.0	0.0
blackscholes	**20.1**	0.0	9.6	76.8	6.9	20.1	0.0	0.0	0.0	0.0	0.0	13.1	0.0	0.0	0.0
swaptions	**42.6**	5.2	36.8	59.2	33.4	2.7	0.0	0.0	35.1	0.0	0.0	0.5	0.0	0.0	0.0
canneal	**2.3**	80.0	81.0	3.6	2.3	5.9	0.0	42.1	96.6	0.0	0.0	47.2	0.0	0.4	0.3
dedup	**3.9**	21.5	27.4	51.3	46.8	0.0	0.0	9.0	42.8	9.5	0.0	21.3	0.0	8.1	9.4
streamcluster	**−14.5**	32.7	31.6	1.5	1.3	0.0	0.0	4.4	50.8	18.2	0.0	67.6	0.0	6.6	12.8

注:表中测试情况 1 为在原有释放同步点的情况下测试程序;测试情况 2 为在情况 1 基础上减少释放同步点数量的情况下测试程序;t_{record}为记录阶段程序的整体运行时间;t_{segv}为程序运行过程中处理页保护异常的总时间;t_{init}为程序运行过程中处理第一次访问共享页面引入的页保护异常的时间;t_{type}为程序运行过程中处理曾经持有又被释放的共享页面引入的页保护异常的时间 $t_{type}=t_{libc}$(或 $t_{request}$ 或 $t_{pthreads}$ 或 $t_{handler}$ 或 $t_{preempted}$);释放同步点包括:libc 类库函数、RequestPermission、Pthreads 类库函数、SIGSEGV 处理函数、同步操作类库函数。

从表 1 可以看出,减少释放同步点的数量的确可以影响并行程序的整体性能:对于部分测试程序来说,性能有所提高,但是对另一部分测试程序来说,性能反而降低,具有相同性能增降趋势的并行程序也并不存在类似的程序特征。另外,对于这两类并行程序而言,处理页保护异常的时间比例有增加,也有减少。在这两种测试情况下,并行程序处理不同原因引入的页保护异常的时间比例也呈非规律性变化。因此,仅仅通过增加或减少释放同步点,对于所有的并行程序并没有统一的效果。

线程释放自己持有的共享页面的访问权限,其效果是作用在两个维度上:①对线程 T_{self} 本身来说,目前持有的某个共享页面一定是曾经访问过,由于程序执行具有局部性,因此后期执行极有可能再次访问这些页面。线程在释放同步点释放所持有的共享页面,导致线程本身在后期执行可能触发额外的页保护异常;②对其他线程 T_{others} 来说,如果该线程申请获得某个共享页面 A 的访问权限时,A 的持有者迟迟不释放,那么 T_{others} 会一直处于阻塞等待状态,降低了并行程序的并行度。最坏情况是,并行程序执行过程中仅有一个线程在执行,其他线程都处于阻塞等待状态。

图 1 所示的程序片段中,共享变量 a、b、c 位于同一个页面 A,线程 x、y、z 分别对变量 a、b、c 赋值之前,均缺少 A 的写访问权限。图 2 给出了该程序片段执行过程中,当提供不同数量的释放同步点时出现的两种典型的执行交叠。

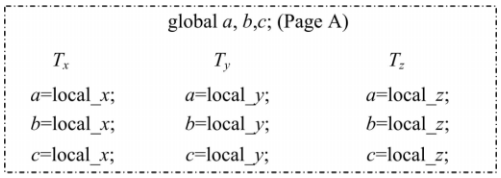

图 1　并行程序片段

Fig. 1　A fragment of parallel program

(1)当出现执行交叠 1 时,线程 x、y、z 同时访问变量 a,由于缺少 A 的写访问权限均触发页保护异常。其中,x 首先获得写访问权限,并顺利对变量 b、c 赋值,线程 y、z 阻塞等待。紧接着,由于线程 x 进入释放同步点释放共享页面 A,相应地,线程 y 在顺利获得对共享页面 A 的写访问权限后,完成对变量 a、b、c 赋值,而此时只有线程 z 仍然阻塞等待。只有当线程 y 进入释放同步点并释放共享页面 A,线程 z 才能继续执行。在执行交叠 1 的情况下,这 3 个线程共触发 3 次页保护异常,但实际上这段时间内真正执行的只有一个线程。

(2)当出现执行交叠 2 时,线程 x、y、z 分别

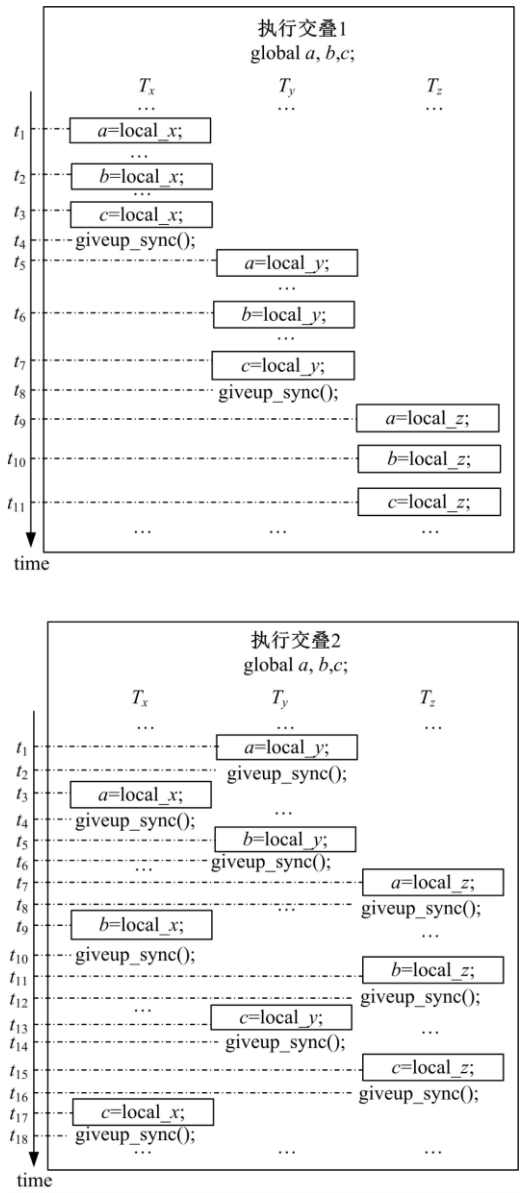

图2 释放同步点对并行程序的影响

Fig. 2 Performance impact for releasing synchronization points

对变量a、b、c的3次赋值操作交叉进行，每次赋值后立即进入释放同步点，将A的访问权限转移给其他等待线程，导致该次执行过程中共触发9次页保护异常。然而，由于每个线程对页面A的持有时间非常短，线程x、y、z交替地访问该页面，与执行交叠1相比，这种情况下引入的页保护异常较多，但程序的并行度较高。从图2和释放同步点的作用效果来看：在缺少同步操作且存在激烈数据竞争的情况下，释放同步点的数量越多，页保护异常的次数越少，程序并行度偏低；相反，释放同步点的数量越少，页保护异常的次数越多，程序并行度偏高。实际上，并行程序的性能是每段时间内各线程执行相互影响的结果，而不是各个效果的简单叠加，无法直观获得并行程序的性能和释放同步点的数量的变化规律。

本文根据并行程序执行过程中的行为特征，提出了一种自适应释放策略，在保持程序较高并行度的前提下，最大限度降低程序处理页保护异常的开销。当释放同步点分散于程序的整个执行轨迹，且线程数目较多时，需要添加更多的同步操作来动态维护元数据信息。在这种情况下，控制开销与自适应释放策略带来的性能收益抵消。因此，不再预设多个释放同步点，只有在线程请求获得访问权限失败时才会动态释放其他线程所需要的共享页面。

2 基于自适应释放策略的低开销确定性重放

2.1 自适应释放

2.1.1 共享页面的释放约束

自适应释放策略的目标是希望选取最合适的线程，在合适的时机释放最合适的共享页面，尽可能改善程序当前的并行度，同时尽量减少自己由于该次释放行为导致后期执行引入的页保护异常。释放线程x与被释放共享页面A的选择需要考虑以下约束条件：

(1)x获取A的历史。线程x目前持有的共享页面，x曾经一定访问过。根据程序执行具有局部性，如果x曾多次获取A，那么后期执行很大可能会再次访问A。如果x获取A的历史次数较多，且每次获取A的时间代价较高，则线程x与共享页面A并不是最优备选。

(2)x持有A的时长。基于公平性考虑，持有时间越长的线程应该及时释放该页面的访问权限。

(3)当前并行度。根据当前各线程的有效执行时间和阻塞时间，判断程序当前的并行度。

(4)释放A后的收益。基于对众多等待线程公平性考虑，释放线程释放对某个共享页面的访问权限时，只有等待队列的队首等待者被唤醒，可能是一个写等待者，也可能是多个读等待者。队首等待者的等待时间，可视为共享页面A被释放后的收益。

2.1.2 释放代价模型

当并行程序中的各个线程均处于阻塞状态时，启动自适应释放，引起各线程阻塞的所有共享

页面暂存于候选集合。为了使每一次共享页面的释放获得的收益最优，综合考虑共享页面访问权限的释放约束，从候选集合中选出最合适释放线程和被释放共享页面移入待释放集合。

为了降低释放代价的计算开销，指定第一个进入阻塞状态的线程作为待释放集合的决策者，其他线程仍处于等待状态。决策线程将为资源公告板上所有被请求的共享页面建立代价模型，计算方式如式(1)所示，由此来量化评估被释放后整个并行程序所能获得的性能收益。

$$P(\mathrm{A}) = \frac{P_{\mathrm{parallel}} \times P_{\mathrm{history}}}{P_{\mathrm{owned}} + P_{\mathrm{wait\_beader}}} \tag{1}$$

式中：P_{owned} 为线程持有该共享页面 A 的时间，$P_{\mathrm{wait\_beader}}$ 为页面 A 的等待队列队首线程的等待时间，如果该共享页面 A 的当前状态为 SHARED_READ，那么等待队列中的各等待线程一定都是请求对 A 的写访问权限，则 $P_{\mathrm{wait\_beader}}$ 是第一个加入等待队列的线程的等待时间，如果 A 的当前状态为 OWNED_WRITE，则 $P_{\mathrm{wait\_beader}}$ 是队首元素为请求写访问权限的线程的等待时间，或队首元素为多个请求读访问权限的线程的等待时间之和；P_{history} 为持有线程在历史执行过程中请求获得 A 的访问权限的时间之和；并行度 P_{parallel} 为当前所有线程执行时间总和占所有线程执行时间与等待时间总和的比例。

根据页保护异常的触发，将线程的执行分割成多个片段。当次 SIGSEGV 触发的开始时刻到当前计算时刻，这段时间称为线程等待时间，从最近一次线程从 SIGSEGV 信号处理函数中返回到此次 SIGSEGV 的触发，这段时间称为执行时间，如图 3 所示。

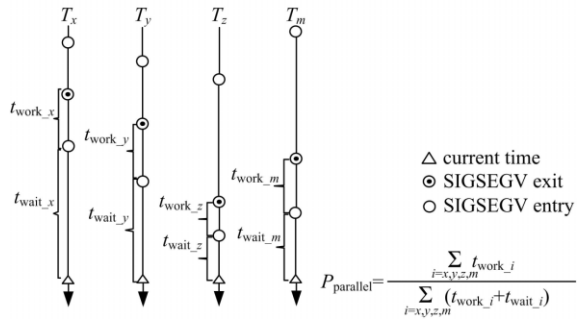

图 3　程序的并行度

Fig. 3　Program parallelism

需要特别说明的是，本文是基于处理器提供的局部时钟来计算上述过程中所涉及的时间。由于目前处理器并不能保证处理器中所有核的局部时钟在同一物理时刻的值一样[18]。因此上述时间都是由各个线程自己利用局部时钟来确定当前的执行时间，例如 X86 架构下的 RDTSC 指令。决策线程根据各线程在各自核上获得的执行时间，为候选集合中每个共享页面计算释放代价。

2.1.3　自适应释放策略

图 4 给出自适应释放算法。其中，释放代价模型由 CalGiveupPriority()函数建立。

Algorithm 1 Adaptive Giving up

```
Input:
  None;
Output:
  releasing set Q;
1:Q<-NULL;//releasing set consisted of pairs of giving up thread &
  gived up page
2:while is Blocking(pWait)==TRUE do
3:    if IsAllThreadsBlocked( ) is true then
4:        LOCK(cal_lock);
5:        if HasCalPriority is not true then
6:            HasCalPriority :=true;
7:            Q:= CalGiveupPriority( ); //calculate cost for
               each registered page
8:        fi
9:        UNLOCK(cal_lock);
10:       for(thread,page) in Q do
11:           if curr_thread == thread then
12:               GiveUpPageAsRequired(page);
13:           else if IsBlockedInFunc(thread) then // giving
              thread is in blocked function
14:               GiveupPageAsPreemped(thread,page);
15:           fi
16:       end for
17:   fi
18: end while
```

图 4　自适应释放算法

Fig. 4　Algorithm of adaptive giving up

线程在执行过程中执行自适应释放的具体步骤如下：

(1)线程在共享页面访问权限请求模块中请求某个共享页面的访问权限失败时，判断当前所有线程是否都处于阻塞状态，如果是，则进入步骤(2)；否则，进入步骤(1)。

(2)第一个进入阻塞状态的线程根据式(1)为候选集合中每个共享页面计算释放代价。释放代价 $P(\mathrm{A})$ 最小的共享页面以及该共享页面的持有者加入待释放集合 Q，进入步骤(3)。

(3)所有线程通过遍历 Q 检查自己是否为释放线程，如果是，则当前线程主动释放相应的共享页面，并记录访问权限的转移情况；否则，进入步骤(4)。

(4)释放线程此时一定阻塞于同步操作类函数的执行,因此当前线程替释放线程释放相应的共享页面,并记录释放信息,以便释放线程后期取消被释放共享页面的访问权限。进入步骤(3),直到集合 Q 遍历完毕。

2.2 库函数重载

为了精准重放并行程序访问共享内存的顺序,需要记录和重放同步操作的顺序,同步操作包括互斥锁、栅栏、条件变量等相关操作。同步操作类库函数的记录和重放根据同步操作的语义进行特殊处理。

(1)互斥锁。截获线程调用 MUTEX_LOCK,在函数出口处记录线程号及抢锁事件。重放阶段时,在 MUTEX_LOCK 入口处,只有当调用线程号、抢锁事件与当前日志项一致时,当前线程才能调用 MUTEX_LOCK,否则阻塞等待。

(2)栅栏。截获线程调用 BARRIER_WAIT,在函数出口处记录线程号及栅栏等待事件。重放阶段时,在 BARRIER_WAIT 入口处,只有当调用线程号、栅栏等待事件与当前日志项一致时,当前线程才能调用 BARRIER_WAIT,否则阻塞等待。

(3)条件变量。图5给出了线程间通过条件变量实现同步的示例。线程 x 在调用 COND_WAIT 后释放用于保护条件变量 cond 的互斥锁 mutex,从而陷入睡眠等待,如图中的箭头①所示。当线程 y 调用 COND_SIGNAL 唤醒等待者线程 x 时,x 会先抢占互斥锁,如图中的箭头⑥所示,再从 COND_WAIT 中返回。为了避免重放条件变量类同步操作时引入死锁,APLAY 首先重载 COND_WAIT,在函数入口处记录线程号及条件变量等待入口事件,在函数出口处记录线程号及条件变量等待出口事件。重放阶段时,

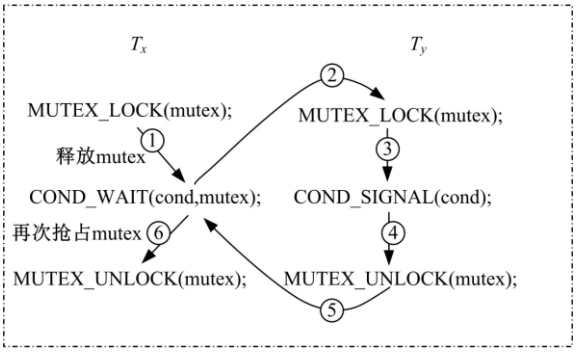

图5 条件变量的使用

Fig. 5 Usage of conditional variable

COND_WAIT 入口处的重放与互斥锁、栅栏类似。重放函数出口时,如果当前日志项与调用线程号不一致,首先释放互斥锁 mutex,然后阻塞等待。直到记录信息与当前调用点匹配,调用线程需再次抢占互斥锁。在这种情况下,仅需保证记录和重放阶段中,最先被条件变量唤醒的线程相同。

2.3 记录/重放

记录/重放模块提供相应的接口供库函数重载、自适应释放调用。记录事件包括:获得共享页面访问权限的顺序、释放共享页面访问权限的顺序、同步操作类库函数的执行顺序。在重放阶段,各线程按序单条读取日志项。对于每一条日志项,每个线程在各重放执行点将其与将要执行的操作逐一匹配,如果匹配失败,则阻塞等待直到读取到正确的日志项。

3 实验评估

为了验证系统的有效性,本文在 Linux 平台上实现了本文提出的基于自适应释放策略的确定性重放系统 APLAY。为了更进一步降低记录开销,APLAY 采纳了 UPLAY 的多种优化策略(包括私有锁、私有堆、主动抢占、进程池等,具体优化算法详见文献[12])。平台参数为:Intel®Xeon®CPU(每个处理器为6核,主频为1.87 GHz),拥有16 G的主存和18 M缓存。操作系统为 Debian 4.4.5,内核版本是 2.6.32,glibc 库版本为 2.11.3。

选用 PARSEC 测试集(2.1 版本)评估 APLAY 系统的性能。PARSEC 测试集是普林斯顿大学提供的并行基准测试程序集[17],覆盖多个科技领域(如计算机视觉、金融、搜索等),具有实际应用的特征。作为多核处理器上共享内存竞争最具代表性的应用程序[19],近年来广泛应用于并行程序研究的学术领域。

通过对比 UPLAY、Scribe 来评估 APLAY 的性能,性能统计对比如图6所示(每个测试程序执行20次)。在本文的实验平台下,Scribe 无法正常运行 streamcluster 测试程序。对比本地执行,Scribe 记录阶段4、8线程的平均降速比为5.50X,9.07X;UPLAY 记录阶段4、8线程的平均降速比为5.64X,9.26X;APLAY 记录阶段4、8线程的平均降速比为4.86X,7.91X。基于多种优化策略[12],UPLAY 的性能与 Scribe 相当,

并且避免了修改操作系统内核；而相比 UPLAY，本文提出的基于自适应释放策略的确定性重放方法使得测试程序的记录性能有了大幅提升。其中，raytrace、canneal 两个测试程序的性能收益最为明显。与其他测试程序不同，这两个测试程序最明显的程序特征是各个线程通过主线程的堆区进行数据交互，且数据伪共享粒度非常细，多个共享页面的访问权限会频繁在线程间转移，并且线程对这些共享页面大部分是写访问；基于 CREW 协议，线程对共享页面写访问权限的申请代价大于读访问。因此，这种数据伪共享引入的页保护异常给并行程序带来非常大的运行时开销。基于对并行程序运行时信息的实时分析，自适应释放算法能够动态确定共享页面访问权限的转移时机，平衡被释放共享页面对该页面持有者本身的代价损耗与对其他阻塞等待线程的代价收益；另外，自适应释放策略对于其他测试程序也有一定的性能提升，例如 ferret、facesim、bodytrack、vips、dedup；由于测试程序 x264、blacksholes、swaptions、streamlcuster 本身各线程间的数据交互较少，或各线程读访问共享页面中的数据更多，故自适应释放与原有预设释放情况下的性能相差不大。因此，相比其他用户态下的确定性重放技术[2,20]（15X-80X 的性能开销），APLAY 能够更进一步地降低确定性重放的记录开销。

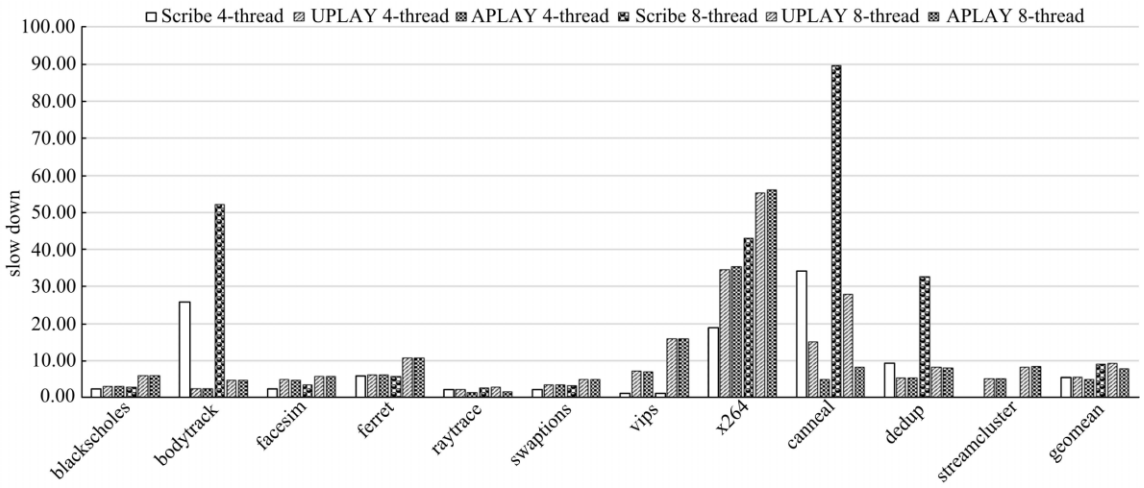

图 6　Scribe、UPLAY 与 APLAY 的性能比较

Fig. 6　Performance comparison of Scribe, UPLAY & APLAY

4　结束语

针对基于页保护异常的确定性重放方法，深入分析和探讨了预设释放同步点对并行程序记录性能的影响；并通过分析并行程序执行过程中的行为特征，定义了共享页面的访问权限释放的约束条件，建立了释放代价模型，从而提出一种基于自适应释放策略的确定性重放方法。实验证明：本文方法能够有效降低并行程序处理页保护异常引入的开销，比其他用户态下的确定性重放技术引入的开销更小。

参考文献：

[1] 蒋炎岩，许畅，马晓星，等. 获取访存依赖：并发程序动态分析基础技术综述[J]. 软件学报，2017，28(4)：747-763.
Jiang Yan-yan, Xu Chang, Ma Xiao-xing, et al. Approaches to obtaining shared memory dependences for dynamic analysis of concurrent programs: a survey[J]. Journal of Software, 2017, 28(4): 747-763.

[2] Patil H, Pereira C, Stallcup M, et al. PinPlay: a framework for deterministic replay and reproducible analysis of parallel programs[C]// The 8th International Symposium on Code Generation and Optimization, Toronto, Ontario, Canada, 2010: 2-11.

[3] Huang J, Liu P, Zhang C. LEAP: lightweight deterministic multi-processor replay of concurrent Java programs[C]// Proceedings of the 8th ACM Sigsoft International Symposium on Foundations of Software Engineering, Santa Fe, NM, USA, 2010: 207-216.

[4] Lee D, Chen P M, Flinn J, et al. Chimera: hybrid program analysis for determinism[C]// Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation,

New York, NY, USA,2012:463-474.

[5] Bond M D, Kulkarni M, Cao M, et al. Efficient deterministic replay of multithreaded executions in a managed language virtual machine[C]// Proceedings of the Principles and Practices of Programming on the Java Platform, New York, NY, USA,2015:90-101.

[6] Honarmand N, Torrellas J. Replay debugging: leveraging record and replay for program debugging [C]// Proceeding of the 41st Annual International Symposium on Computer Architecuture, Minneapolis, MN, USA, 2014:445-456.

[7] Pokam G, Danne K, Pereira C. QuickRec: prototyping an intel architecture extension for record and replay of multithreaded programs[C]// Proceedings of the 40th Annual International Symposium on Computer Architecture, Tel-Aviv, Israel, 2013:643-654.

[8] Honarmand N, Dautenhahn N, Torrellas J, et al. Cyrus: unintrusive application-level record-replay for replay parallelism[C]// Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems, Houston, TX, USA, 2013:193-206.

[9] Honarmand N, Torrellas J. Replay debugging: leveraging record and replay for program debugging [C]// Proceeding of the 41st Annual International Symposium on Computer Architecuture, Minneapolis, MN, USA, 2014:445-456.

[10] Pokam G, Pereira C, Hu S L, et al. CoreRacer: a practical memory race recorder for multicore x86 TSO processors[C]// Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture, Porto Alegre, Brazil, 2011:216-225.

[11] Laadan O, Viennot N, Nieh J. Transparent, lightweight application execution replay on commodity multiprocessor operating systems[C]// Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, New York, USA, 2010:155-166.

[12] **应欢,王东辉,武成岗,等. 适用于商用系统环境的低开销确定性重放技术[J]. 吉林大学学报:工学版**,2017,47(1):208-217.
Ying Huan, Wang Dong-hui, Wu Cheng-gang, et al. Efficient deterministic replay technique on commodity system environment[J]. Journal of Jilin University(Engineering and Technology Edition),2017, 47(1):208-217.

[13] Wang Z, Li J, Wu C, et al. HSPT: practical implementation and efficient management of embedded shadow page tables for cross-ISA system virtual machines[C]// Proceedings of the 11th ACM Sigplan/Sigops International Conference on Virtual Execution Environments, Istanbul, 2015:53-64.

[14] Kedia P. Efficient deterministic replay through dynamic binary translation[D]. Delhi: Indian Institute of Technology, 2015.

[15] Lee D. Holistic system design for deterministic replay[D]. Ann Arbor: University of Michigan, 2013.

[16] Honarmand N. record and deterministic replay of parallel programs on multiprocessors[D]. Urbana: University of Illinois, 2014.

[17] Bienia C, Kumar S, Singh J P, et al. The PARSEC benchmark suite: characterization and architectural implications[C]// Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques, Toronto, Ontario, Canada, 2008:72-81.

[18] Yuan X, Wu C, Wang Z, et al. ReCBuLC: reproducing concurrency bugs using local clocks[C]// Proceedings of the 37th International Conference on Software Engineering, Florence, 2015:824-834.

[19] **唐士斌. 多线程程序的确定性调试方法[D]. 北京: 中国科学院计算技术研究所**,2014.
Tang Shi-bin. Debugging muiltithreaded programs with determinism[D]. Beijing: The Institute of Computing Technology , Chinese Academy of Sciences, 2014.

[20] **陈宇飞. 可伸缩的确定性重放技术研究[D]. 上海: 复旦大学计算机科学技术学院**,2014.
Chen Yu-fei. Improving the scalability of deterministic replay[D]. Shanghai: School of Computer Science, Fudan University, 2014.