计算机研究与发展
Journal of Computer Research and Development
DOI:10.7544/issn1000-1239.2015.20150160
52(6):1266-1277,2015

一种多线程程序内存系统模拟器 Trace 驱动仿真方法

朱鹏飞[1,3]　卢天越[2,3]　陈明宇[2]

[1](计算机体系结构国家重点实验室(中国科学院计算技术研究所)　北京　100190)

[2](中国科学院计算技术研究所先进计算机系统研究中心　北京　100190)

[3](中国科学院大学　北京　100049)

(zhupengfei@ict.ac.cn)

A Trace-Driven Simulation of Memory System in Multithread Applications

Zhu Pengfei[1,3], Lu Tianyue[2,3], and Chen Mingyu[2]

[1](*State Key Laboratory of Computer Architecture* (*Institute of Computing Technology*, *Chinese Academy of Sciences*), *Beijing* 100190)

[2](*Center for Advanced Computing Systems*, *Institute of Computing Technology*, *Chinese Academy of Sciences*, *Beijing* 100190)

[3](*University of Chinese Academy of Sciences*, *Beijing* 100049)

Abstract　Nowadays, chip-multiprocessors (CMPs) become significantly important for multithread applications due to their high-throughput performance in big data computing. But growing latency to memory is increasingly impacting system performance because of memory wall. Two independent simulation methods: trace-driven and execution-driven, are available for system researchers to study and evaluate the memory system. On one hand, in order to leverage simulation speed, researchers employ trace-driven simulation because it removes data processing and is faster than execution-driven counterpart. On the other hand, lack of data processing induces both global and local trace misplacements, which never exist in multithread applications on real machine. Through analytical modeling, remarkable performance metrics variations are observed due to trace misplacements. Basically speaking, the reasons are in trace-driven simulation: 1) locks do not prevent threads from non-exclusively entering critical range; 2) barriers do not synchronize threads as need; 3) the dependence among memory operations is violated. In order to improve memory system simulation accuracy in multithread applications, a methodology is designed to eliminate both global and local trace misplacement in trace-driven simulation. As shown in experiments, eliminating global trace misplacement of memory operation induces up to 10.22% reduction in various *IPC* metrics, while eliminating local trace misplacement of memory operation induces at least 50% reduction in arithmetic mean of *IPC* metrics. The proposed methodology ensures multithread application's invariability in trace-driven simulation.

Key words　trace-driven simulation; accuracy; memory system; multithread applications; trace collection and replay

摘　要　伴随大数据计算时代的到来,片上多核处理器为提高多线程程序服务器吞吐率发挥巨大作用,同时其内存系统的访问延迟越来越影响系统性能.目前,路径驱动(trace-driven)仿真方法比执行驱动(execution-driven)运行速度快,被内存系统研究者广泛采用.但是路径驱动在仿真并发线程时,会同时导致宏观和微观的访存错位.而实际多线程程序运行过程中,不会发生这种访存错位行为.通过理论分

收稿日期:2015-02-16;**修回日期**:2015-04-22

基金项目:国家自然科学基金项目(61272132,61221062)

析和计算,访存错位引起路径驱动的仿真结果存在明显偏差.针对上述问题,提出了一种方法来避免路径驱动仿真发生宏观和微观访存错位,精确回放采集阶段的多线程程序行为.实验数据显示,在避免宏观访存 trace 错位后,多线程程序的多个仿真指标出现最高 10.22%的变化;对于部分访存密集型的多线程程序,避免微观访存 trace 错位可以使算数平均 IPC 出现大于 50%的变化.为研究交互线程的内存系统行为提供一种更加准确的路径驱动方法.

关键词 路径驱动仿真;精确度;内存系统;多线程程序; trace 采集回放

中图法分类号 TP391.9

近年来,随着大数据计算时代的到来,片上多核处理器集成了越来越多的运算处理单元,作为主要共享资源的内存系统面临着越来越大的数据交互需求.计算机体系结构研究显示 cache 层次、内存调度[1]、DRAM 结构等内存系统上的综合优化能有效提高系统处理大数据的并发能力.由于用真实硬件运行标准测试程序(benchmark)的方法很难评估新设计的内存系统,研究早期甚至不可能通过系统原型的方法来实验内存设计中的每个创新点,因此研究人员广泛采用模拟器仿真的方法来探索内存系统设计空间.如何快速建立内存系统准确的仿真模型成为需要解决的问题.

为仿真片上多核处理器,执行驱动(execution-driven)模拟器的规模越来越复杂,并且随着片上处理器核数增加,数据集合以及数据访问量都在急剧增加.执行驱动仿真器运行整体系统,不得不详细设计和模拟每个功能单元.特别地,巨大的数据集和请求数目会导致执行驱动仿真效率低.与执行驱动模拟器不同,路径驱动(trace-driven)模拟器[2]通过回放原始程序的执行过程进行仿真.由于不需要进行真正数据计算以及数据搬移,路径驱动模拟器运行速度快.路径驱动能快速建模,给研究者较大的灵活度来实验不同的设计参数.而且,通过从不同指令集真实机器上获得的内存访问 trace,路径驱动仿真方法能够接受异构机器[3-4]的仿真,可以对比研究内存系统(比如调度算法、控制器设计以及内存新器件等)在不同机器上的性能,实现同一个仿真器研究不同指令集机器.

但是,在研究多线程应用的内存系统性能时,路径驱动仿真方法存在局限性.第一个局限性是路径驱动模拟器不能建模多线程程序的并行线程间(inter-thread)行为,比如线程间的序和同步.而内存系统研究的目的之一是调度仲裁多线程的内存请求[5],解决来自多个处理器核的内存访问间冲突问题,从而提高共享内存系统的性能和公平性.当用路径驱动模拟器回放多线程内存 trace 来仿真内存系统时,由于输入 trace 中只有地址,没有可用来判断线程行为的数据值,仿真会出现以下 2 个宏观上的问题:1)仿真中交替进入关键区进行处理的线程行为,与真实机器收集 trace 时的线程行为不一致;2)仿真中栅障(barrier)函数无法起到真正强制多线性同步的作用.在这 2 种情况中,模拟器回放的 trace 片段发生宏观错位.路径驱动仿真的另一个局限是缺少必要信息进行访存操作依赖的检查.通常在执行驱动的仿真中包含操作数的寄存器依赖检查;而在 trace 仿真驱动中,为了简化 trace 文件,仅仅保存每个访存操作的主要内存信息,比如访存地址、数据大小,而没有操作数的寄存器信息.但是,访存地址是通过地址寄存器运算而得到的,寄存器的依赖会引起访存操作之间的依赖.如果不能正确保证依赖关系,单个线程的访存操作之间就会出现微观上的错位,同样误导内存系统设计.

本文的目标是设计一种仿真方法来维护路径驱动仿真中线程间的行为,确定性地回放 trace 采集阶段的线程行为,避免宏观访存错位;同时用访存 trace 的执行顺序保证微观上每个线程正确的访存操作依赖关系,从而提高用路径驱动仿真方法研究并行多线程应用下内存系统时的准确性.

根据实验数据(参见第 3 节),采用本文的方法避免宏观访存 trace 错位后,加权 IPC(instruction per cycle)、调和平均 IPC 以及 CPI(cycle per instruction)吞吐量等多个仿真指标会出现大约 10%的变化;而避免微观 trace 错位会引起部分访存密集型程序平均 IPC 指标出现大于 50%的变化;综合避免宏观和微观访存错位,多种内存系统指标会出现 10%~20%的变化.

1 研究动机

1.1 Trace 驱动仿真中的宏观 trace 错位

宏观访存 trace 错位指仿真回放时多线程获得

锁(lock)的顺序发生错误或者无效的栅障函数,引起某个线程比其他线程运行过快或者过慢.出现宏观错位的 trace 片段,会引起错位的内存请求之间在处理器仿真时存在不真实的干扰/冲突.例如内存系统接受来自多线程程序的访存操作就是不符合实际程序执行的行为,cache 层次、缓存一致性协议、内存调度算法、内存通道、DRAM 行缓冲器、DRAM 页等共享资源受多个处理器核之间的"错位干扰/冲突"影响,甚至多个"错位"trace 片段显示的线程行为是一种在真实机器中并不存在的混乱行为.结果,路径驱动模拟器仿真多线程应用环境下内存系统就会产生有偏差的性能指标,误导内存系统设计.

以多线程程序大量使用"锁"和"栅障"为例.这 2 种工具可以保护关键区以及实现同步.在任何时候,只有一个线程能够成功获得"锁"、执行关键区代码,而其他所有试图获得同一个"锁"的线程都要等待前个线程释放掉"锁".因此,在 trace 收集阶段得到的多线程程序关键区代码访存 trace 片段,反映的是线程成功获得"锁"的顺序一致.图 1(a)显示了多线程中常见的一种"锁"的关系:生产者与消费者关系.从 thread1 到 thread0 传送数据,需要通过共享结构实现,为防止破坏共享,插入和删除操作需要关键区保护.消费者线程 thread0 总是要等生产者线程 thread1 填充列表之后才能获取数据.thread0 在 t_0 时刻之间进入关键区但没有可用数据,然后 thread1 在 t_2 进入关键区填充数据到 t_4 时刻释放锁之后,thread0 再次进入关键区取得数据,然后 2 个线程开始新的处理过程.路径驱动模拟器回放上述过程中的关键区 trace 时,由于 trace 中不含数据计算,模拟器无法判断列表的空满,仅仅盲目地向前运行 trace,会出现宏观访存 trace 错位,导致不能准确回放线程间的交互行为.例如,仿真中如果 thread0 比 thread1 运行得快,thread0 就会获得锁并且运行 2 遍关键区 trace,而 thread1 这时还没进入关键区向列表中填充数据.这种消费者先于生产者得到数据的线程间交互行为不会在真实机器中发生.从内存系统角度来看,真实机器中 trace 片段 A 必须先于 thread0 第 2 次进入关键区且 trace 片段 B 应该晚于 thread0 第 2 次进入关键区,在这个时间阶段来访问内存;而 thread0 两次进入关键区前后的访存不能同 thread1 进入关键区的访问同时运行.但仿真中,由于 tread0 比 tread1 运行得快,这种竞争不存在了,会使仿真趋向好的结果.更进一步,宏观 trace 错位会不断累积,引起后续 trace 片段越来越多的错位,影响仿真结果准确度.如果模拟器能够确保回放时线程获得锁的顺序同收集阶段的顺序相同,线程间交互关系就是正确的,就能够使内存系统在真实准确的工作环境下仿真,多个线程对内存共享资源的访问是有效的.

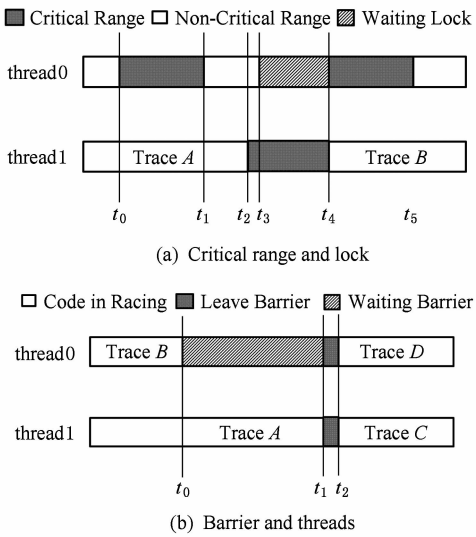


Fig. 1 Trace alignment controlled by lock and barrier.

图 1 锁和栅障控制下 trace 对齐方式

多线程程序中"栅障"会出现与"锁"类似的情况,导致宏观访存错位.在真实机器运行多线程程序时,任何到达"栅障"的线程都应该被阻塞,等待其他线程都到达同一个"栅障",这个行为暗示执行"栅障"前后的 trace 片段之间有先后顺序.用路径驱动仿真回放时,"栅障"前的任一段 trace 不能晚于"栅障"后的任何 trace.图 1(b)显示了 2 个线程"栅障"同步的例子.图 1(b)中,晚于"栅障"的 trace 片段 D 必须等 thread1 到达"栅障"后才能运行,即必须在 trace 片段 A 执行结束后;否则,仿真中线程失去了同步关系.对于内存系统而言,仿真时只能出现 trace A 与 trace B 或者 trace C 与 trace D 之间同时访问内存系统的情况,而不可能出现 trace A 与 trace D 之间同时访问的情况.

1.2 Trace 驱动仿真中的微观 trace 错位

微观访存 trace 错位指仿真回放中单个线程有依赖关系的 2 个访存操作执行顺序和执行时刻违反依赖关系.违反依赖关系的访存操作,在实际系统中不能发生,这就产生了仿真误差,

寄存器依赖是指在计算机程序指令序列中,后面指令的操作数使用了与前面指令操作数相同的寄存器.处理器设计中存在 3 种寄存器依赖:读后写、写后读和写后写.其中读后写和写后写是假依赖,通

过寄存器重命名能够实现动态调度,寄存器之间的执行关系序不会影响程序结果;但是写后读是真相关,表示后面指令的操作数依赖前面指令的结果,在动态调度乱序执行处理器中无法通过寄存器重命名的方法解决寄存器写后读的依赖关系,必须严格地按照先后顺序执行,这就影响系统性能.通常执行驱动仿真包含有操作数的寄存器依赖检查,而 trace 驱动仿真没有办法进行模拟.原因在于 trace 驱动仿真需要事先保存多线程程序的访存 trace 文件,为了简化 trace 文件,仅仅保存每个访存操作的主要内存信息,没有操作数的寄存器信息,也不能为每条指令都保存大量的寄存器信息.但是,访存地址是通过地址寄存器运算而得到的,表 1 所示为 Intel 指令集格式和指令中关于内存操作的地址计算方法.

Table 1 Intel Instruction Format and Addressing Method

表 1 Intel 指令集格式和访存操作地址计算

Instruction	Example	Addressing
Read Memory	mov 0x8 (% rbx,% rax, 1), %rdx	%rbx+%rax×1+8
Write Memory	mov %rcx,0x0(%rbp,%rcx, 8)	%rbp+%rcx×8+0
Read and Write Memory	add %rdx,(%rbx,%rax,1)	%rbx+%rax×1
Non-Memory	mov %rax,%rbp	

每条访存都对应一个基址(base)寄存器和一个索引(index)寄存器计算访存地址,访存指令的地址寄存器之间的依赖会引起访存操作之间的相关.图 2 是遍历链表计算链表中元素个数的源代码和汇编代码.从图 2 我们可以看出,每次遍历链表中一个元素需要 4 条指令,其中第 1 条指令是一个访存读指令,遍历每个元素时访存读指令之间是相互依赖的,不能独立并发执行.假设需要遍历含 1 000 个元素的链表,根据文献[6]的间隔模型公式,在一个含两级缓存层次的内存系统上,乱序调度处理器执行上述所有指令的总时钟数为

$$C=\sum_k \frac{N_k}{D}+m_{iL1}\times c_{iL1}+m_{iL2}\times c_{iL2}+m_{br}\times(c_{dr}+c_{fe})+m_{dL2}\times c_{dL2},$$

其中,N_k 是第 k 次流水线停顿前连续提交的指令条数;D 是处理器的发射窗口宽度(N_k 并不总是 D 的整数倍);m_{iL1},m_{iL2},m_{br} 和 m_{dL2} 分别是发生一级、二级指令缓存失效的指令条数、发生分支预测错误的指令条数和不可重叠的二级数据缓存失效的指令条数;c_{iL1},c_{iL2},$(c_{dr}+c_{fe})$ 和 c_{dL2} 分别是上述这些指令造成的延迟.假设遍历链表操作的所有指令在指令缓存 100%命中(即 $m_{iL1}=0$,$m_{iL2}=0$),分支预测器是 100%准确的(即 $m_{br}=0$),但是有 1%的链表元素发生数据缓存失效,需要进行访存,所有访存之间是有依赖关系,不能被重叠执行.以发射窗口为 6,访存的延迟是 250 个处理时钟周期计算,全部的执行时间就是 $C=1\,000\times4/6+1\%\times1\,000\times250=3\,166$,则 CPI 和 IPC 分别是 3 166/4 000=0.79 和 4 000/3 166=1.26.如果我们在路径驱动仿真中不能保证访存指令之间的依赖关系,则变成相互独立可重叠的内存操作,这样只需要计算第一个访存的延迟,其他访存因为并发执行分担和掩盖了延迟,全部执行时间就是 $C=1\,000\times4/6+1\times250=916$,这时 CPI 和 IPC 变成 0.23 和 4.36,带来的指标误差是 3 倍.

```
node_t * current=head;
int count=0;                          4004e7:48 8b 7f 08   mov  0x8(%rdi),%rdi
while (current! =NULL){               4004eb:83 c0 01      add  $0x1,%eax
        count++;                      4004ee:48 85 ff      test %rdi,%rdi
        current=current→next;         4004f1:75 f4         jne  4004e7<Length+0x7>
}
```

Fig. 2 Loops to calculate the number of items in linked-list.

图 2 遍历链表计算元素个数

通过以上分析可见,用 trace 驱动方法研究多线程程序内存系统性能时,需要模拟器确保回放阶段线程间交互行为和同步行为与 trace 收集阶段相同,同时根据访存操作之间的依赖关系控制操作执行的时刻和顺序,来保证内存系统接收的访存请求是真实准确的,确保内存访问间干扰和冲突是真实准确的.

2 仿真方法改进

本文提出一种精确模拟多线程程序内存系统的路径驱动仿真方法,避免仿真多线程应用时发生宏观和微观访存 trace 错位.针对宏观访存 trace 错位,

在 trace 收集阶段从多线程程序中选择一些关键插桩点,截获程序执行过程中线程获得锁的顺序和线程在栅障函数作用下的同步行为,将锁的顺序和同步关系记录到 trace 中;接下来,在 trace 回放仿真阶段,模拟器精确重建线程获得锁的顺序和栅障函数的同步行为.针对微观访存 trace 错位,在 trace 收集阶段通过维护寄存器和访存操作的映射关系,把寄存器依赖关系转换成访存操作依赖关系保存在 trace 中;在 trace 回放仿真阶段,模拟器检查访存操作之间依赖关系决定访存操作的执行时刻和顺序.这种仿真方法结合插桩技术和路径驱动仿真,只用少量关键信息就可以保证 trace 收集和回放 2 个阶段宏观上线程间序和同步的一致性,微观上线程内访存操作的依赖关系不变,提高用路径驱动方法研究多线程应用下内存系统的准确性.

仿真方法包含 3 个关键手段,确保在 trace 收集阶段和 trace 回放阶段访存操作关系一致:1)在多线程程序中选择位置进行插桩,目的是识别第 1 节中描述的 2 类程序执行点,即"锁"的执行点和"栅障"的执行点.在这些执行点处,用插桩的方法把线程间宏观的交互顺序标记到 trace 中.2)对每条指令插桩,通过分析寄存器依赖关系映射出访存操作依赖关系,根据资源窗口的限制,把一定范围内的访存依赖关系记录到 trace 文件中.3)利用标记的线程交互信息和访存操作依赖关系,在内存系统模拟器中精确回放 trace 片段.精确回放的目的是消除仿真中可能出现的宏观和微观访存 trace 错位,同时根据线程的交互行为向仿真模型注入合适的动态 trace,进一步保证宏观执行正确性和准确性.

图 3 显示这种仿真方法的基本流程,包括 2 个主要步骤:1)用插桩方法收集 trace,需要重点监控程序动态指令流中"锁"和"栅障"的执行点以及寄存器依赖关系.①将线程交互顺序和同步关系标记到 trace 片段中;②分析寄存器依赖以及其与访存操作的对应关系,把访存操作间的依赖记录到 trace 中.2)读取标记有线程同步关系和访存依赖关系的 trace 片段进行 trace 驱动仿真,模拟器负责处理线程在标记点处的交互行为以及保证线程内访存操作的正确依赖关系;仿真过程中取预先处理过的 trace 模板作为输入,当线程到达同步标记点处时模拟器读取 trace 模板,参考所标记的线程关系要求生成动态 trace 片段补充到原有的 trace 中.在这种仿真方法中,模拟器不仅避免了由于宏观访存 trace 错位而引起的多线程关键区执行混序和同步问题,也消除了微观访存 trace 错位带来的访存依赖关系混乱,保证内存系统在一种同原始执行过程相同的环境中进行仿真.下面详细介绍上述 2 个关键手段的实现方法.

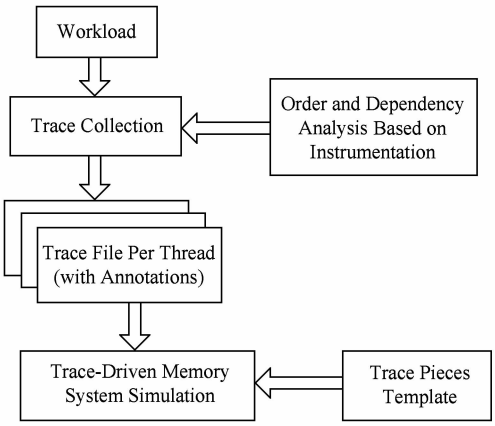


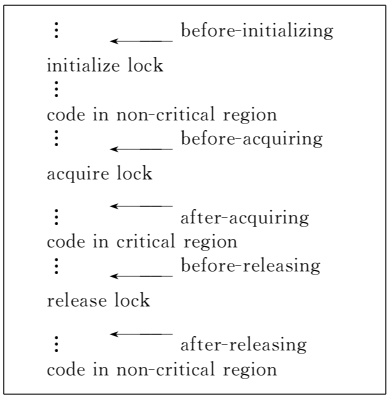
Fig. 3 Simulation methodology.

图 3 仿真方法框图

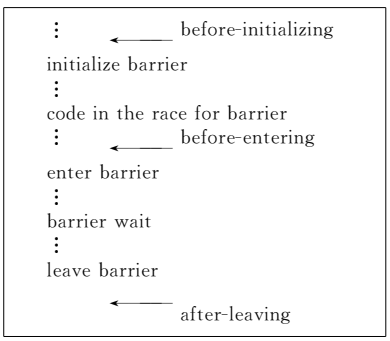
2.1 插桩位置

借助编译技术,我们能够静态地或者动态地向应用程序的任何位置插桩额外代码.本文提出的仿真方法充分利用插桩技术的优势,寻找多线程程序的关键执行点,将执行中重要的线程关系标记到 trace 中.针对宏观访存 trace 错位(线程关键区交互混乱和线程同步问题),有 2 类重要的执行点能够反映所需宏观信息,即"锁"的执行点和"栅障"的执行点.

图 4 显示标记上述 2 类执行点的具体插桩位置.每个"锁"操作对应 2 个"锁"的执行点:获取"锁"和释放"锁",每个执行点有一个入口和一个出口,则记录"锁"行为总共需要 4 个插桩位置:获取之前、获取之后、释放之前和释放之后,每个插桩位置反映不同的线程关系.获取之前表示仿真中的本线程可能获取"锁"成功从而结束等待;而获取之后表示仿真中本线程确定性地成功获取"锁"并结束等待.相对应地,释放之前表示仿真中其他某个线程可能获取"锁"成功从而结束等待;而释放之后表示其他某个线程确定性地成功获取"锁"并结束等待."栅障"操作就是阻塞,所以每个"栅障"有 2 个插桩位置:进入阻塞之前和离开阻塞之后.进入阻塞之前表示如果本线程比其他线程运行得快,可能要被阻塞在"栅障"处;而离开阻塞之后表示所有线程都确定性地开始新的运行.除此以外,"锁"和"栅障"都需要在初始化之前的位置插桩来准备 trace 收集中用到的数据结构.



(a) Critical region



(b) Barrier

Fig. 4 Instrumentation points.

图 4 需要插桩的关键点

对上述执行点进行插桩,除了收集线程交互关系和同步关系外,还要去除每对插桩点之间的 trace 片段.这部分去除的 trace 片段,在回放中可以通过 trace 模板以及标记的信息,重新被动态注入到模拟器.

在"锁"的执行点,插桩函数为每个"锁"维护一个全局共享计数器.根据插桩点反映线程关系的不同,插桩函数向 trace 中标记同步信息.获得锁之前,插桩函数负责标记"锁"号(即锁地址),并且停止收集本线程 trace;获得锁之后,插桩函数把这个"锁"对应的计数器值标记到 trace 中,表示本线程获得锁的全局顺序号,然后重新开始收集本线程 trace.释放锁之前插桩函数再次把"锁"号标记到 trace 中,并停止收集本线程 trace,还要自增"锁"的计数器,将自增后的计数器值也标记到 trace 中,表示下一个可以获得锁的线程全局顺序号.此处插桩的自增计数器的操作,仍处于本线程关键区执行过程中,因而不需要额外"锁"保护.释放锁之后插桩函数不需要记录信息,只需要重新开始收集本线程 trace.

在"栅障"执行点,插桩函数为每个"栅障"维护一个变量,变量值就是在这个"栅障"处同步的线程个数.进入阻塞之前,插桩函数同时将"栅障"号(即"栅障"地址)连同所有需要同步的线程个数一起标记到 trace 中,然后停止收集本线程 trace;离开阻塞之后,插桩函数开始重新收集本线程 trace.整个过程滤除插桩点之间的 trace,而仅仅记录线程同步信息.图5给出一个例子,pthread程序经过插桩后标

```
int thread_func()
{
  ⋮
  //code segment A
  ⋮
  pthread_barrier_wait(&bar);
  //code segment B
  ⋮
  pthread_spin_lock(&lock);
  //code in critical region
  ⋮
  pthread_spin_unlock(&lock);
  //code segment D
  ⋮
  pthread_barrier_wait(&bar);
  //code segment E
  ⋮
}
```

(a) pthread lock/unlock/barrier

⋮	⋮	⋮
NonMem	5	
RD	2B	Load. addr
WR	2B	Store. addr
BarWait	2	Bar. addr
⋮	⋮	⋮
NonMem	5	
WR	2B	Store. addr
⋮	⋮	⋮
LockAcq	No. 2	Lock. addr
⋮	⋮	⋮
RD	2B	Load. addr
⋮	⋮	⋮
LockRls	No. 3	Lock. addr
⋮	⋮	⋮
RD	4B	Load. addr
⋮	⋮	⋮
BarWait	2	Bar. addr
NonMem	5	
RD	2B	Load. addr
⋮	⋮	⋮

(b) thread0 trace

⋮	⋮	⋮
NonMem	5	
RD	2B	Load. addr
WR	2B	Store. addr
BarWait	2	Bar. addr
⋮	⋮	⋮
NonMem	5	
WR	2B	Store. addr
⋮	⋮	⋮
LockAcq	No. 3	Lock. addr
⋮	⋮	⋮
RD	2B	Load. addr
⋮	⋮	⋮
LockRls	No. 4	Lock. addr
⋮	⋮	⋮
RD	4B	Load. addr
⋮	⋮	⋮
BarWait	2	Bar. addr
NonMem	5	
RD	2B	Load. addr
⋮	⋮	⋮

(c) thread1 trace

Fig. 5 Trace examples with lock and barrier.

图 5 带锁和栅障的 trace 片段

记有“锁”和“栅障”信息的 trace 片段.通常,trace 中每项记录含 3 个内容:操作类型、数据大小和内存地址.针对特殊标记项,操作类型表示“锁”或“栅障”执行点的类型,比如获得锁、释放锁和栅障阻塞;数据大小表示“锁”计数器值或者“栅障”同步的线程个数;内存地址表示“锁”地址或者“栅障”地址,即“锁”号或“栅障”号.

2.2 访存依赖关系

仿真中存在 2 种访存操作依赖关系:访存地址依赖和写访存数据依赖.1)访存地址依赖指由寄存器真相关(写后读)引起、后面访存操作的访存地址寄存器直接(或间接)依赖于前面访存操作的结果寄存器.访存地址依赖要求后面访存操作必须等前面被依赖的访存操作执行结束才能执行访存,例如进行一级缓存的访问.2)写访存数据依赖指由寄存器真相关(写后读)引起、后面写访存操作的数据寄存器直接(或间接)依赖于前面访存操作的结果寄存器.写访存数据依赖关系中的写访存操作必须等待前面被依赖的访存操作执行结束后才能执行.在这 2 种访存操作依赖中,访存地址依赖是真相关,无法通过重命名的方法解决,在仿真中必须限制访存操作的执行时刻;而写访存数据依赖虽然也是一个真相关,但通过延迟写操作的方法,可让写访存提前提交,真正的写操作等待数据就绪后再进行,不影响处理器性能.然而,从研究内存系统仿真的角度来看,实际的写操作仍然是在被相关的访存操作结束后执行的.在包含缓存层次的内存系统仿真中,需要限制写操作的执行时刻,等待前一个被相关的访存结束再执行,来反映访存的正确行为.

图 6 为指令序列(来自 HPCC[7] 测试集之一的 LCG 关键循环).汇编代码执行产生的 trace 文件包含 5 个访存操作(4 个读操作、1 个写操作).如果不考虑访存操作数依赖,trace 驱动仿真会将这 5 个操作连续发射到内存系统并发执行他们.但是,通过寄存器依赖分析,第 2 个访存读操作的地址依赖第 1 个读操作的访存结果(即依赖前一个访存用 dep.1 表示),同样第 4 个读操作地址依赖第 3 个读操作的访存结果,而第 5 个访存操作的写数据间接依赖第 4 个访存操作的访存结果(通过非访存指令 xor).所以,这些访存操作的正确执行顺序应当是第 2 个访存操作等待第 1 个访存操作结束,第 4 个访存操作等待第 3 个访存操作结束,第 5 个访存操作等待第 4 个访存操作结束.

C source codes: *Table*[*ran*[*j*]≫(64−*logTableSize*)]^=*ran*[*j*];	
ASM codes:	Trace file:
shl $0x3,%rax	NonMem
add −8416(%rbp),%rax	RD 8B Load.addr0 dep.0
mov (%rax),%rdx	RD 8B Load.addr1 dep.1
mov −28(%rbp),%eax	RD 4B Load.addr2 dep.0
cltq	NonMem
mov −8272(%rbp,%rax,8),%rax	RD 8B Load.addr3 dep.1
xor %rdx,%rax	NonMem
mov %rax,(%rsi)	WR 8B Store.addr4 dep.1

Fig. 6 Memory operation dependence and trace file.
图 6 访存操作依赖关系指令序列

为了在 trace 驱动的仿真中实现 2 种访存依赖,需要将访存操作之间的依赖关系加入 trace 中.通过插桩每条指令,用映射表方式分析维护寄存器与访存操作之间的对应关系,记录每个寄存器依赖第几个访存操作.插桩函数更新映射表的过程为:执行非访存操作时,将源寄存器依赖的访存操作编号作为目标寄存器依赖的访存操作编号,直接更新映射表中目标寄存器项;执行访存操作时,将地址寄存器依赖的访存操作编号更新目标寄存器项.由于可能存在多个依赖,我们简单地选择距离最近的访存操作作为被依赖对象,这是为了简化插桩函数、缩短插桩后应用程序的执行时间.本文并不记录所有访存操作的精确时间序,而是在生成访存操作 trace 时除了记录访存地址和数据大小等必要信息还记录访存依赖关系:从所有地址寄存器中选择最近的访存操作作为依赖.对于写操作,检查写数据寄存器的依赖,选择最近的操作.为了减少 trace 文件大小,这里并不需要记录被依赖操作的绝对编号,只需要记录本操作和被依赖操作的间隔个数.因为现代处理器的资源深度是有限的,比如发射窗口、rob 缓冲大小以及缓存失效状态保持寄存器(MSHR)的大小.这些资源限制了依赖的操作不能距离太远,否则可以认为被依赖访存操作已经结束,所以我们只需要增加有限位就可以记录依赖关系.

2.3 精确仿真 trace

在回放的时候避免宏观和微观访存 trace 错位,trace 驱动的仿真器必须根据标记在 trace 中的锁序号、同步标记以及访存依赖关系控制 trace 的回放过程.针对宏观访存 trace 错位,即保证锁和同步的正确回放顺序,模拟器需要建立一个内部的计数器和一个表.这个表用来跟踪锁顺序和栅障同步,表中记录当前系统中多线程程序锁和栅障的状态,

比如哪个线程得到锁或者哪些线程已经到达栅障.本文提出多线程仿真过程如下:开始时刻,仿真器按照正常模式运行每个线程的 trace,直到某个线程遇到特殊标记,比如遇到请求锁操作、释放锁操作或者栅障操作,这时仿真器读取内部映射表,用仿真当前的线程状态比较路径文件中记录的标记信息,决定是否允许线程继续执行路径文件中后续指令.换句话说,当仿真状态和路径文件中标记的线程状态匹配,才能继续执行.例如,如果某个线程正在执行获取锁的操作,路径文件中标记这个线程是第几个得到锁的线程,然后仿真器看当前计数器是否是这个线程进入关键区的正确顺序,如果一致就可以继续执行关键区,否则等待直到内部计数器轮到这个线程的正确顺序.

为避免微观访存 trace 错位,仿真器要检查有依赖关系访存操作的发射和提交,根据依赖关系动态控制每个操作进入内存系统的时刻.比如,如果被依赖的访存操作没有完成,后面相关的访存操作不能访问 cache,保证了访存操作之间的正确执行关系.

完成上述控制后,当某个线程因为宏观顺序需要等待时,仿真器不能简单停止线程仿真,需要继续产生等待操作,这部分操作在路径采集阶段被滤掉了,如果不恢复就会对系统功耗的仿真产生较大影响.因为在真正的系统中,等待过程中的操作会额外消耗系统能量,可用预先建立 trace 片段模板的方法恢复程序访存行为.对于不同的同步函数,可以选择注入不同的行为,例如 spinlock 的行为就是不断访问同一个地址直到成功;而 barrier 是等待系统唤醒.这样一来,在线程等待正确的宏观顺序时,仿真器能够正确再现多线程程序执行阶段线程的行为.

3 实验评估

为评估第 2 节方法避免宏观和微观 trace 错位后 trace 驱动仿真器结果的变化,本文以研究 4 种内存调度算法(FR-FCFS[1],ATLAS[8],STFM[9],PA-BS[5])为例,选择多个包含锁和栅障操作的典型多线程测试程序(Parsec[10],Graph500-BFS[11],STREAM[12],HPCC[7],SSCA[13]),比较内存系统仿真结果变化.实验仿真平台配置如表 2 所示,为增加仿真准确性采用自编的 trace 驱动时钟精确仿真器模拟完整的片上多核处理器结构及其内存系统(包括多发射乱序执行处理器、两级缓存层次、缓存一致性协议、片上网络以及主存控制器),其中主存控制器使用 DRAMSim2 模拟[14].首先,按照本文第 2 节介绍的方法使用 Pin[15-16]编写 trace 文件的采集工具,收集上述多线程程序在真实机器上执行的 trace 文件,每个多线程程序并发 8 个线程;然后用上述仿真平台执行多线程程序 trace.第 1 次仿真时,模拟器不考虑第 1 节提到的关键区乱序和同步问题,并且允许有依赖关系的访存操作并发执行;而后续多次仿真,使用本文提出的回放方法,运行有代表性的多线程程序比较仿真结果,分别保证线程之间正确的宏观同步锁的操作(避免宏观 trace 错位)和微观访存操作依赖关系(避免微观 trace 错位).实验选取第 1 次的结果为基准,后续结果归一化到基准结果上进行比较.

Table 2 Configuration of Experiment Platform

表 2 仿真器配置

Unit	Configuration
Processor	8 cores, 4 GHz, out-of-order issue, 128 instruction window, 4 issue/4 commit (one memory operation) per cycle, 16 MSHR.
Cache Hierarchy	L1 private cache, 16 KB, 4-way set associative, 64 B line. Shared L2 cache, 4 MB, 16-way set associative, 64 B line, directory based cache-coherent protocol: MESI.
Network on Chip	2x4 mesh network, one router per node.
Memory Controller	2 on-chip memory controllers, 32 entries transaction queue, 32 entries command queue, FR-FCFS, open page, rank-interleave address mapping.
Main Memory	16 GB main memory, Micron device: DDR3-1333 MHz, x8, 8banks, 32768 rows/bank, 1024 columns/row, 1 KB page size, $BL=8$.

为了显示宏观 trace 错位对各种内存调度算法都有影响,本文选择 Parsec 测试程序比较避免宏观 trace 错位前后 4 种内存调度算法(FR-FCFS,ATLAS,STFM,PA-BS)仿真结果的变化.图 7 显示避免了宏观 trace 错位后,4 种内存调度算法的仿真结果存在 10%左右变化,其中所有调度算法的 3 种指标(加权 *IPC*、调和平均 *IPC* 以及 *CPI* 吞吐量)变化的几何平均值分别是 10.22%,9.30%和 10.22%.

为了显示微观 trace 错位对内存系统仿真的影响,本文选择访存密集型多线程程序 HPCC,STREAM,BFS,SSCA 比较避免微观 trace 错位后 4 种内存调度算法(FR-FCFS,ATLAS,STFM,PA-BS)仿真结果的变化.图 8 显示避免微观访存错位后部分访存密集型多线程程序,在 4 种内存调度算法条件下 *IPC* 变化 50%左右,说明这些程序本身包

含有数据访问的依赖,严格限制有依赖关系的访存操作执行时刻和顺序,仿真指标发生较大变化.图 8 中,depe 表示有微观错位,nodepe 表示避免微观错位.

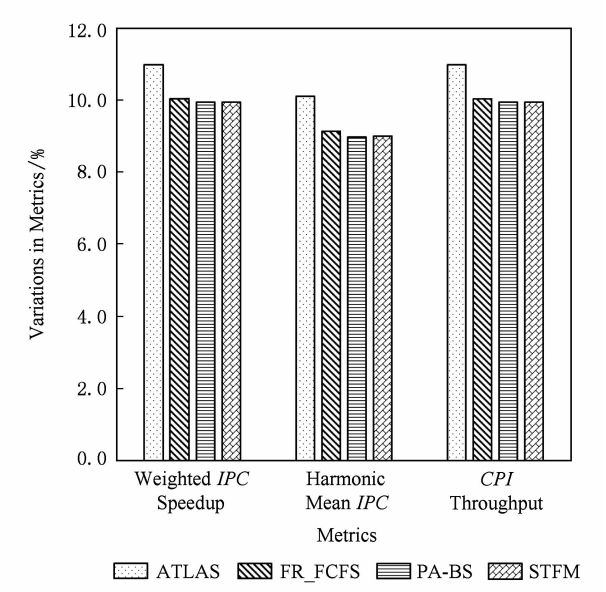


Fig. 7 Effects of avoiding global trace misplacement.

图 7 避免宏观 trace 错位后仿真结果的变化

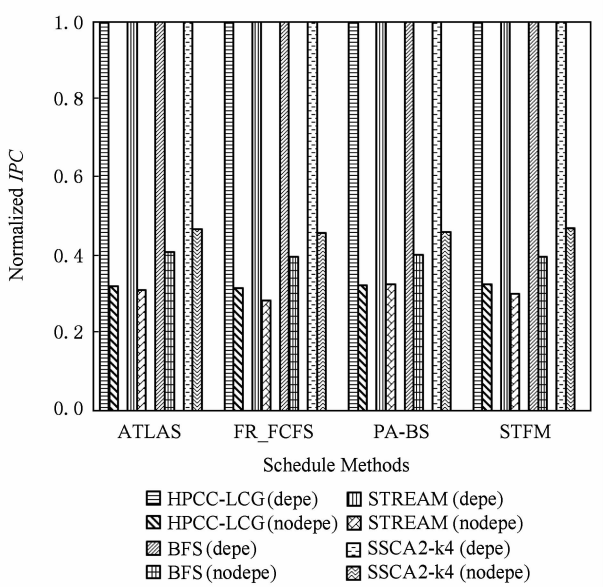


Fig. 8 Effects of avoiding local trace misplacement.

图 8 避免微观 trace 错位后仿真结果变化

综合宏观和微观 trace 错位,以常见的 FR-FCFS 内存调度算法运行 Parsec 多线程程序为例,图 9 显示同时避免宏观和微观访存错位后,处理器性能指标(算数平均 *IPC*)和 2 种内存系统性能指标(行缓冲器命中率(row buffer hit rate)和 bank 并行度(level parallel))的变化.从图 9(a)可见,宏观访存错位对 blacksholes 和 swaptions 的影响不大,原因是这 2 个多线程程序没有用到锁和同步的操作,基本可以看作多个程序单独执行的行为.其中 bodytrack 变化较大:避免宏观错位后,bodytrack 仿真结果 *IPC* 减少 20%,原因是 bodytrack 程序中用到许多锁和同步的操作.从图 9(b)(c)的结果看,消除宏观访存错位后,bodytrack 的行缓冲命中率变低,锁和同步作用消除了错位访存引起的命中率.图 9 显示微观访存错位对 swaptions 和 fluidanimate 的影响较大,

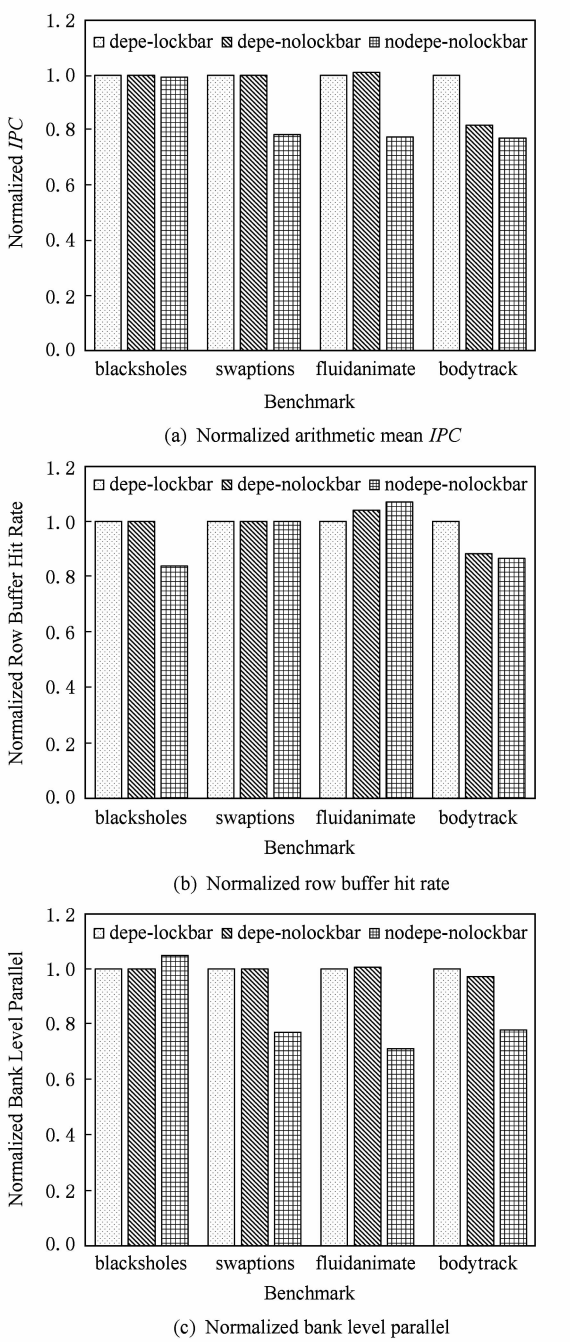


Fig. 9 Effects of avoiding both global and local trace misplacement.

图 9 避免宏观和微观访存错位后系统仿真指标变化

分别引起 *IPC* 减少 21%和 25%;而对其他 2 个多线程应用影响不大,总体变化小于 5%.在内存系统指标方面,除 blacksholes 和 bodytrack 外,微观错位对行命中率影响不大;但除 blaksholes 外,微观错位对 bank 并行度影响大于 20%.图 9 中,depe-lockbar 表示有宏观和微观访存错位;depe-nolockbar 表示避免宏观访存错位、有微观访存错位;nodepe-nolockbar 表示避免宏观和微观访存错位.

实验表明通过本文的方法,多线程程序路径驱动仿真避免宏观和微观访存错位,可以使各种仿真结果出现 10%~20%的变化,结合第 1 节理论分析,本文的仿真能够精确回放 trace 采集阶段的多线程行为,使路径驱动仿真结果更加准确.

4 相关工作

近几年来,正由于本文第 1 节所描述的特点,内存系统研究使用多种基于路径驱动方法的仿真器.其中由马里兰大学(University of Maryland)开发的 DRAMSim2 是时钟精确内存模拟器[14],它能够准确地模拟 DDR2 或者 DDR3 内存系统模块.通过设定不同的内存性能参数,DRAMSim2 模拟了每一个时钟周期内内存系统各个部件的运行情况和状态更新,对内存系统进行了较为精确的模拟,被广泛采用.DRAMSim2 仅仅是主存控制器的模拟,没有考虑多处理器的线程间同步和访存操作数相关的情况,需要结合其他处理器端仿真器才能使用.USIMM[17]是由犹他大学开发的时钟精确内存模拟器来模拟 DRAM 的结构和时序.USIMM 的内存系统控制器包含 Write queue(写队列,提高连续的写内存命令的效率)和多内存功耗模式控制等模拟技术.为了提高模拟器的准确性,USIMM 除了用 Re-order Buffer 模块模拟 CPU 发射指令的限制外,还考虑内存系统响应速度对 CPU 的影响,但仍然没有考虑线程间交互的情况.CMU 大学研究内存系统的文献[18-19]采用了自编的路径驱动仿真器,没有显示使用了线程内微观的操作数依赖.本文采用的模拟器在 DRAMSim2 的基础上设计了完整的片上多核系统,对多级缓存层次、缓存一致性协议和片上网络的模拟增加了仿真平台的完整性和准确性.

部分内存系统研究[5,20]采用路径驱动的仿真方法集中研究多个应用程序环境下的内存控制器调度算法,而非针对多线程程序.多核系统在运行多个不同应用程序时,不需要准确考虑程序间相互的执行顺序和同步,因此,路径驱动仿真成为首选手段.文献[21]选择合适的策略能够对多个程序的主存操作进行优先级排序,可以消除对共享资源比如控制器和总线的干扰和冲突.新的排序实际就是对原始路径片段的错位,这种错位准确反映了真实系统在应用调度策略时取得的效果.但是,多线程应用程序本身的存储操作之间具有内在约束,不能任意调整线程之间访存的宏观执行顺序.文献[19,22]用路径驱动仿真方法研究并行多线程程序的内存调度算法,但未提到如何防止访存错位.

文献[23]在研究对线程应用的路径驱动仿真方法中,采用了一种协同操作机制.这种机制同时组合了路径驱动和执行驱动 2 种仿真方法,能够在调度并行程序的多个线程时切换 2 种仿真方法.在主存调度研究领域,广泛使用采样仿真方法[24-25],选择提取具有代表性的访存片段进行仿真.这时线程的数目总是固定等于系统中的处理核个数,没有动态调度线程仿真的必要.文献[23]中的方法不是针对精确回放 trace,和本文解决 trace 错位的问题不同.

目前部分多线程程序的研究是用动态二进制插桩工具(或者功能仿真器)把在线 trace 动态地注入研究对象的仿真模型中,如 CMP $ im[26].另外一个例子是 COTSon[27],通过 AMD 的功能仿真引擎把功能仿真和时序仿真分开.与纯路径驱动仿真比较,这种方法虽然节约了用来存储路径文件的磁盘空间,但是因为每次仿真中都需要同时运行应用程序和仿真器,会额外耗费仿真时间,这和本文提出的将 trace 采集和 trace 回放分成 2 步不同.

不确定性是实际系统执行多线程程序时常见的现象[28],实现确定性回放[29-31]是多线程程序仿真的一个重要方向,应用之一是调试程序或者验证系统.在不同情况下,线程可能以不同的顺序得到锁,从而出现多线程程序的非确定性.这种不确定性会引起仿真指标(例如 *IPC*)的变化,不利于对比不同系统结构的性能[28].但是实现确定性回放会在执行中占用额外主存空间,并且需要保存大量的程序执行信息.本文提出的确定性 trace 驱动仿真方法,目标是避免 trace 驱动仿真中的访存 trace 错位,通过记录线程执行关键点和线程内数据依赖的关系序而非精确的时间序,实现在仿真时每 2 个执行关键点之间程序片段按照采集时相同的顺序执行,间接消除了多线程程序仿真回放的不确定性,实现确定性回放采集时线程执行关系,为 trace 驱动方法研究多线程交互行为下的内存系统提供一种可能性.

访存错位会对内存调度算法产生影响,文献[32]仅仅研究了锁和同步的作用,并没有同时针对宏观和微观错位进行有效工作.本文同时避免微观和宏观错位,提出更加准确的仿真方法提高整个内存系统的试验准确度.文献[33]的 trace 中记录有非访存指令的依赖关系,同时文献[33]的仿真器没有缓存层次,依靠采集阶段得到的每个访存失效情况来控制 trace 回放时到达主存的访存操作.本文的仿真方法包含完整片上多核系统及其缓存层次,在采集阶段通过非访存指令传递相关链,只记录访存指令之间的依赖关系,同时结合窗口深度简化 trace 收集和回放过程,有利于保证仿真速度的同时提高精度.

5 未来工作

本文提出的方法适用于多线程程序内存系统仿真,要求多线程程序使用明确的接口,例如使用 pthread 标准库中的 splinlock/mutex.目前,有些多线程应用直接通过编译器进行并行化,比如使用 openMP,插桩技术虽然能够找到这些操作的接口,但需要我们仔细分析编译后的代码,找出同步点的函数位置.有的多线程应用程序使用了自定义的锁和同步方式,例如使用了 pthread 的条件变量,就需要研究者额外的工作,分析程序行为提取出操作接口进行正确的插桩.这些是下一步的研究方向.本文通过理论分析结合实验证明了 trace 错位的影响,下一步工作是结合真实系统的运行结果,对比分析改进后路径驱动仿真方法同执行驱动仿真方法的准确度差别.

6 结束语

本文重点提出解决路径驱动方法研究多线程程序内存系统时的局限性,即如何避免宏观和微观访存 trace 错位.本文提出通过少量额外信息,在 trace 文件中同时标记多个线程宏观获得锁的顺序和到达栅障的同步信息以及访存操作的微观依赖关系,并在仿真中严格按照标记信息确定性回放 trace 文件.理论分析和实验表明我们的方法能够提高内存系统仿真的准确度.

最后,本文提出的方法为研究多线程程序内存系统提供了一种更加准确的模型.未来,通过对更多更复杂的执行点插桩,能够更加深入研究和评估多线程程序内存系统.

参 考 文 献

[1] Rixner S, Dally W, Kapasi U, et al. Memory access scheduling [C] //Proc of the 27th Annual Int Symp on Computer Architecture. New York: ACM, 2000: 128-138

[2] Uhlig R, Mudge T. Trace-driven memory simulation: A survey [J]. ACM Computer Survey, 1997, 29(2): 128-170

[3] Joao J, Suleman M, Mutlu O, et al. Bottleneck identification and scheduling in multithreaded applications [C] //Proc of the 17th Int Conf on Architectural Support for Programming Languages and Operating Systems. New York: ACM, 2012: 223-234

[4] Kayiran O, Nachiappan N, Jog A, et al. Managing GPU concurrency in heterogeneous architectures [C] //Proc of the 47th Annual IEEE/ACM Int Symp on Microarchitecture. Los Alamitos, CA: IEEE Computer Society, 2014: 114-126

[5] Mutlu O, Moscibroda T. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems [C] //Proc of the 35th Annual Int Symp on Computer Architecture. Los Alamitos, CA: IEEE Computer Society, 2008: 63-74

[6] Eeckhout L. Computer Architecture Performance Evaluation Methods, Synthesis Lectures on Computer Architecture [M]. San Rafael, CA: Morgan& Claypool Publishers, 2010

[7] Vetter J. Contemporary High Performance Computing: From Petascale toward Exascale [M]. Boca Raton, FL: Chapman & Hall/CRC, 2013

[8] Kim Y, Han Dongsu, Mutlu O, et al. ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers [C] //Proc of the 16th Int Symp on High Performance Computer Architecture. Piscataway, NJ: IEEE, 2010: 1-12

[9] Mutlu O, Moscibroda T. Stall-time fair memory access scheduling for chip multiprocessors [C] //Proc of the 40th Annual IEEE/ACM Int Symp on Microarchitecture. Los Alamitos, CA: IEEE Computer Society, 2007

[10] Bienia C, Kumar S, Singh J, et al. The PARSEC benchmark suite: Characterization and architectural implications [C] //Proc of the 17th Int Conf on Parallel Architectures and Compilation Techniques. New York: ACM, 2008: 72-81

[11] Ueno K, Suzumura T. Highly scalable graph search for the Graph500 benchmark [C] //Proc of the 21st Int Symp on High-Performance Parallel and Distributed Computing. New York: ACM, 2012: 149-160

[12] McCalpin J. STREAM: Sustainable memory bandwidth in high performance computers [R]. Charlottesville, VA: University of Virginia, 2007

[13] Gilbert J, Reinhardt S, Shah V. High performance graph algorithms from parallel sparse matrices [C] //Proc of the 8th Int Conf on Applied Parallel Computing: State of the Art in Scientific Computing. Berlin: Springer, 2006: 260-269

[14] Rosenfeld P, Cooper-Balis E, Jacob B. Dramsim2: A cycle accurate memory system simulator [J]. Computer Architecture Letters, 2011, 10(1): 16-19

[15] Lueck G, Patil H, Pereira C. PinADX: An interface for customizable debugging with dynamic instrumentation [C] // Proc of the 10th Int Symp on Code Generation and Optimization. New York: ACM, 2012: 114-123

[16] Luk C, Cohn R, Muth R, et al. Pin: Building customized program analysis tools with dynamic instrumentation [C] // Proc of the 2005 ACM SIGPLAN Conf on Programming Language Design and Implementation. New York: ACM, 2005: 190-200

[17] Chatterjee N, Balasubramonian R, Shevgoor M, et al. USIMM: The Utah SImulated Memory Module A Simulation Infrastructure for the JWAC Memory Scheduling Championship [R]. Salt Lake City, Utah: The University of Utah, 2012

[18] Yoon H, Meza J, Muralimanohar N, et al. Efficient data mapping and buffering techniques for multi-level cell phase-change memories [J]. ACM Trans Architecture Code Optimization, 2014, 11(4): 40:1-40:25

[19] Zhao Jishen, Mutlu O, Xie Yuan. FIRM: Fair and high-performance memory control for persistent memory systems [C] //Proc of the 47th Annual IEEE/ACM Int Symp on Microarchitecture. Los Alamitos, CA: IEEE Computer Society, 2014: 153-165

[20] Nesbit K, Aggarwal N, Laudon J, et al. Fair queuing memory systems [C] //Proc of the 39th Annual IEEE/ACM Int Symp on Microarchitecture. Los Alamitos, CA: IEEE Computer Society, 2006: 208-222

[21] Cheng H Y, Lin C H, Li J, et al. Memory latency reduction via thread throttling [C] //Proc of the 43rd Annual IEEE/ACM Int Symp on Microarchitecture. Los Alamitos, CA: IEEE Computer Society, 2010: 53-64

[22] Ebrahimi E, Miftakhutdinov R, Fallin C, et al. Parallel application memory scheduling [C] //Proc of the 44th Annual IEEE/ACM Int Symp on Microarchitecture. New York: ACM, 2011: 362-373

[23] Rico A, Duran A, Cabarcas F, et al. Trace-driven simulation of multithreaded applications [C] //Proc of 2011 IEEE Int Symp on Performance Analysis of Systems and Software. Piscataway, NJ: IEEE, 2011: 30-37

[24] Sherwood T, Perelman E, Hamerly G, et al. Automatically characterizing large scale program behavior [C] //Proc of the 10th Int Conf on Architectural Support for Programming Languages and Operating Systems. New York: ACM, 2002: 45-54

[25] Patil H, Cohn R, Charney M, et al. Pinpointing representative portions of large Intel Itanium programs with dynamic instrumentation [C] //Proc of the 37th Annual IEEE/ACM Int Symp on Microarchitecture. Los Alamitos, CA: IEEE Computer Society, 2004: 81-92

[26] Jaleel A, Cohn R, Luk C K, et al. CMP $ im: A Pin-based on-the-fly multi-core cache simulator [C] //Proc of the 4th Annual Workshop on Modeling, Benchmarking and Simulation. Los Alamitos, CA: IEEE Computer Society, 2008 [2014-12-15]. http://citeseerx. ist. psu. edu/viewdoc/summary?doi=10. 1. 1. 143. 5627

[27] Argollo E, Falcon A, Faraboschi P, et al. COTSon: Infrastructure for full system simulation [J]. SIGOPS Operating Systems Review, 2009, 43(1): 52-61

[28] Alameldeen A, Wood D. Variability in architectural simulations of multi-threaded workloads [C] //Proc of the 9th Int Symp on High-Performance Computer Architecture. Los Alamitos, CA: IEEE Computer Society, 2003: 7

[29] Lu Kai, Zhou Xu, Wang Xiaoping, et al. An efficient and flexible deterministic framework for multithreaded programs [J]. Journal of Computer Science and Technology, 2015, 30(1): 42-56

[30] Chen Yunji, Hu Weiwu, Chen Tianshi, et al. LReplay: A pending period based deterministic replay scheme [C] //Proc of the 37th Annual Int Symp on Computer Architecture. New York: ACM, 2010: 187-197

[31] Zheng Long, Liao Xiaofei, Wu Song, et al. A replay system for performance analysis of multi-threaded programs [J]. Journal of Computer Research and Development, 2015, 52(1): 45-55 (in Chinese)
(郑龙, 廖小飞, 吴松, 等. 一种用于多线程程序性能分析的重放系统[J]. 计算机研究与发展, 2015, 52(1): 45-55)

[32] Zhu Pengfei, Chen Mingyu, Bao Yungang, et al. Trace-driven simulation of memory system scheduling in multithread application [C] //Proc of the 2012 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness. New York: ACM, 2012: 30-37

[33] Lu Tianyue, Chen Licheng, Chen Mingyu. An optimization on memory system simulator based on trace accuracy improvement [C] //Proc of the 20th National Conf of Information Storage. Beijing: China Computer Federation, 2014 (in Chinese)
(卢天越, 陈荔城, 陈明宇. 一种基于 Trace 精度改进的内存系统模拟器优化方法[C] //第 20 届全国信息存储技术学术会议. 北京: 计算机学会, 2014)

Zhu Pengfei, born in 1978. PhD candidate. His current research interests mainly include computer architecture.

Lu Tianyue, born in 1992. PhD candidate. His current research interests mainly include computer architecture (lutianyue@ict. ac. cn).

Chen Mingyu, born in 1972. Professor and PhD supervisor. His current research interests include architecture, operating system and algorithm optimization for high performance computing (cmy@ict. ac. cn).