MODV 存储一致性模型验证工具的性能优化①

赵晓凯[1,2], 孙鲁明[1,3]

[1](中国科学院软件研究所, 北京 100190)
[2](中国科学院大学, 北京 100190)
[3](中国银行数据中心, 北京 100094)

摘 要: MODV 是一个通用的存储一致性模型动态验证工具, 该工具实现了基于时间序的边界图算法, 具有较低的时间复杂度. 为了进一步提高 MODV 工具的性能, 我们采用了多种方法对算法进行了性能优化, 使得 MODV 工具能够有效验证更大规模的并发访存操作. 实验结果表明, 和基准算法相比, 我们的改进算法在性能方面有较大的提升.

关键词: 存储一致性模型; 动态验证工具; 边界图算法; 时间序; 性能优化

Performance Enhancement of MODV Memory Consistency Model Verifitcation Tool

ZHAO Xiao-Kai[1,2], SUN Lu-Ming[1,3]

[1](Institute of Software, Chinese Academy of Sciences, Beijing 100190, China)
[2](University of Chinese Academy of Sciences, Beijing 100190, China)
[3](Data Center, Bank of China, Beijing 100094, China)

Abstract: MODV is a general purpose dynamic verification tool for memory consistency model. It implemented a time order based frontier algorithm, which has the lower time complexity. To achieve higher performance of MODV, we design and implement several accelerate strategies for its algorithm. The experimental results show that the performance of enhanced algorithm is better than the baseline algorithm.

Key words: memory consistency model; dynamic verification tool; frontier graph algorithm; time order; performance enhancement

1 引言

存储一致性模型向程序员提供了内存系统的形式化描述, 消除了程序员对系统的预期行为与系统真实行为之间的差距[1]. 最早的内存一致性模型是由 Lamport 提出的顺序一致性模型(sequential consistency)[2]. 顺序一致性模型容易被程序员所理解, 但它对系统的执行要求过于严格, 阻碍了很多优化措施. 随着互联网的普及, 人们对计算机系统的性能要求越来越高, 多核处理器系统得到了广泛应用. 为了提高系统的性能, 多核处理器系统一般会使用弱存储一致性模型(relaxed memory model). 在弱存储一致性模型下运行程序时, 处理器不必严格按照程序的顺序执行. 内存操作一般分为普通读写操作和同步操作, 带同步操作的弱存储一致性模型被称为同步一致性模型(synchronized consistency model). 同步一致性模型在一些用于研究领域的多核处理器系统中得到了应用[3,4]. 在这类系统中, 通常使用同步操作来限定程序中指令的执行次序, 因此程序的行为会更加复杂, 给验证工作带来了很大的困难.

由于对存储一致性模型进行验证有着重要的实际意义, 因此学术界和工业界都有着广泛的研究. Gibbons 和 Korach 将顺序一致性模型的验证问题简化为 VSC-read 问题, 并通过将其转换成 3SAT 问题, 证明了对顺序一致性模型的验证是 NP 完全问题[5]. 由此

① 收稿时间:2015-03-03;收到修改稿时间:2015-04-17

可见, 验证大规模系统的存储一致性模型是一个极具挑战的问题. 对存储一致性模型的验证可以分为静态验证和动态验证两类方法. 静态验证用形式化语言描述内存系统的设计及存储一致性模型的性质, 通过定理证明或者模型检测的方式来验证系统是否满足存储一致性模型. Condon 和 Park 等人使用静态验证方法对此问题进行了研究[6,7]. 这种方法的优点在于可以在设计层面尽可能多地找到系统存在的诸多问题, 缺点在于无法对大规模系统进行验证.

动态验证在工业界有着广泛应用. 动态验证首先在实现存储一致性模型的处理器下实际执行测试用例, 然后通过检查运行结果, 对存储一致性模型进行验证. 其优点在于, 可以通过程序的运行过程及运行结果获取更多的信息, 利用这些信息, 降低算法的时间复杂度. 缺点在于, 验证结果会受到测试用例覆盖度的影响, 并且对验证程序有所限制. 动态验证又可以分为基于硬件辅助的方法和基于软件的方法两类. 在基于硬件辅助的方法中, 可以通过辅助硬件获取读操作与写操作之间的对应关系, 以及访问同一地址的写操作之间的执行顺序, 因此可以将验证算法的时间复杂度降低到 O(n)[8], 其中, n 是程序中内存操作的数目. 虽然这种方法的验证时间较少, 但是由于需要辅助硬件支持, 一方面会给系统设计带来负担, 另一方面会降低系统的性能.

基于软件的动态验证不需要辅助硬件的支持, 常被用于流片后阶段的验证. 由于在短时间内需要完成大规模并发程序的验证, 这类方法或工具的性能是工具有效性的决定性因素. Gibbons 等人提出了基于边界图(frontier graph)的验证算法, 用于验证顺序一致性模型[5], 时间复杂度为 $O(n^p)$, 其中 p 为处理器的数目. Hangal 等人设计并实现了 TSOtool 工具, 可以对顺序一致性模型和写全序模型进行验证[9]. TSOtool 是第一个可以对实际的多核处理器系统进行存储一致性模型验证的工具, 其设计思路影响了后来的许多研究. 该算法的时间复杂度为 $O(n^5)$. Manovit 等人通过在 TSOtool 中引入向量时钟(vector clock), 将算法的时间复杂度降到 $O(p*n^3)$[10]. Intel 公司的 Roy 等人提出了基于回溯的算法, 时间复杂度为 $O(n^4)$[11], 其验证工具被集成到 Intel 的硬件验证平台, 能够验证更多的存储一致性模型. 之后, Hangal 等人对自己之前提出的算法进行了改进, 使其达到完备性, 时间复杂度为 $O(n^p/p^p*p*n^3)$[12]. 上述工作的不足之处有两点, 一是由于时间复杂度是多项式级别, 在程序规模变大时, 无法进行有效的验证; 二是验证范围局限在顺序一致性模型和写全序模型, 无法对同步一致性模型进行验证.

Chen 等人设计并实现了 LCHECK 工具, 其中引入了时间序的概念, 对既可靠又完备的验证, 时间复杂度降低至 $O(C^p*p^2*n^2)$, 对可靠但不完备性的验证, 时间复杂度降低至 $O(p^3*n)$ [13]. 随后, Hu 等人设计并实现了 XCHECK 工具, 在使用时间序的基础上, 利用 Gibbons 所提出的边界图算法, 将时间复杂度降低至 O(n)[14]. 虽然通过时间序可以有效降低算法的时间复杂度, 大大提升工具的性能, 但是时间序仅仅适用于对顺序一致性模型的验证, 对于弱存储一致性模型, 如果直接使用时间序, 可能会得出错误的验证结果.

上述的研究工作主要从公理语义的角度进行验证, Burnim 等人从操作语义的角度出发, 对 TSO 及 PSO 内存一致性模型下执行是否满足顺序一致性进行了研究, 他们给出一种新颖的监视算法, 使得验证达到了可靠性和完备性[15]. 另外, 自 TSOtool 起的一系列工具都是通过检测某次执行是否满足相应的内存一致性模型, 他们对算法的优化集中在验证算法本身, Chen 等人提出了新的思路[16], 他们通过对待验证的程序进行正则化, 来提升验证工具的性能. Alglave 等人提出了一种方法, 通过对测试程序进行变换, 将对弱一致性模型的验证问题转换为对顺序一致性模型的验证[17]. 另外, 以上的工作都是基于 CPU 的, 而近些年来, 在大规模计算方面, GPU 得到了广泛使用, Sorensen 等人对基于 GPU 的内存一致性模型的验证进行了研究[18].

2 MODV结构和算法

由于之前的研究工作对同步一致性模型的验证少有涉及, 因此, 我们针对同步一致性模型, 开发了通用的存储一致性模型验证工具 MODV[19,20], 支持对各种弱存储一致性模型进行验证, 同时改进了时间序的概念, 既保证了算法的正确性, 又有效地降低了算法的时间复杂度.

MODV 由三个模块组成, MODV 随机指令生成模块, 操作执行窗口推断模块和 MODV 分析模块, 如图 1 所示. 其中操作执行窗口推断模块不属于标准 MODV 系统. 由于它和多核处理器系统底层相关, 所

以在实际验证时, 需要根据具体的多核处理器系统, 来决定使用何种方式得到操作的执行窗口.

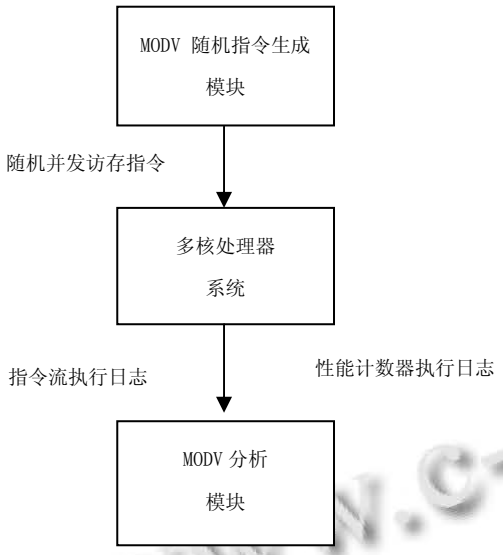

图1 MODV系统结构

MODV 随机指令生成模块根据用户定义的地址数目, 操作比例, 随机产生访存指令流. MODV 分析模块读取指令流执行序列及指令执行窗口信息, 并根据存储一致性模型的约束函数和指令窗口信息, 判断程序的执行结果是否与存储一致性模型一致[20].

MODV 验证算法的核心思想是根据待验证模型的公理语义, 定义程序中操作之间的序关系, 根据这个序关系, 为操作之间加相应的边, 最终构成一个有向图, 验证算法检测图中是否有环, 若无环, 该程序满足模型, 否则, 验证算法即找到一个反例.

我们首先对边界图算法[5]进行描述. 假设需要验证的程序共 n 个内存操作, 运行在 p 个处理器上. 每一个边界 f 都是由 p 个操作构成的集合, 即 $f = (u_1,u_2,...,u_p)$, 每个元素 u_i 对应处理器 i 中的一个操作. 根据 Jentsen 不等式, 边界图中最多有 $O(n^p/p^p)$ 个节点. 我们将开始边界定义为包含 p 个空操作的集合, 结束边界为包含每个处理器中的最后一个内存操作的集合. 根据存储一致性模型的公理语义确定相邻边界转换规则.

算法的基本思想为: 从开始边界出发, 相邻边界只改变一个元素, 并且不能违反相邻边界转换规则, 如果最终能够到达结束边界, 那么程序满足待验证的存储一致性模型, 否则程序不满足该模型. 下面是算法描述:

算法 1 : 边界图算法

Input: 程序在某个存储一致性模型下的执行结果和该模型的公理语义约束

Output: 如果可以到达结束边界, 返回 true; 否则返回 false

① 在操作之间添加由存储一致性模型的公理语义规定的静态边;

② if 加边后有环 then

③ return false;

④ $f_0 \leftarrow$ 开始边界;

⑤ sat $\leftarrow$ ExploreActiveFrontier(f_0);

⑥ return sat;

验证算法如算法 1 所示. 算法首先为程序中的内存操作之间添加静态边, 然后检查图中是否存在环. 由于静态边表示操作之间确定的序关系, 所以如果存在环, 边界图算法一定无法到达结束边界, 即程序的运行结果不满足待验证的存储一致性模型. 否则, 使用递归函数 ExploreActiveFrontier 不断尝试寻找下一边界, 根据算法最终能否到达结束边界, 返回算法的最终验证结果.

ExploreActiveFrontier 函数在寻找下一边界时, 需要符合相邻边界转换规则. 是否符合相邻边界转换规则, 由同步一致性模型的语义决定. 根据同步一致性模型的语义, 程序中访问同一地址的写操作和使用同一把锁的同步操作可以通过所加的动态边形成一个全序. 这里, 我们认为模型要求所有的写操作都要有全序.

ExploreActiveFrontier 的运行过程是, 从边界 f 开始, 通过深度优先遍历的方式遍历所有可能的边界, 在遍历过程中, 每次扩展新边界时, 为边界的有效操作之间加动态边, 不同类型的有效操作有不同类型的动态边, 如果在加边的过程中形成环, 则说明这一次对边界的扩展不符合相邻边界转换规则, 需要回溯访问下一个可能的边界. 如果最后能到达结束边界, 返回 true, 否则返回 false.

在函数 ExploreActiveFrontier 中, 替换边界 f 的下一个边界为 f′. 由于 f′中只有一个元素与 f 中不同, 所以 f′的可能性非常多, 这会形成大量的无效边界, 最终造成反复回溯. 为了降低时间复杂度, 提高算法的效率, 我们引入了时间序的概念. 时间序利用了

Lamport 在并发程序中所提出的全局时钟概念[21]. 如果一个操作的结束时间在另一个操作的开始时间之前, 那么前面的操作一定可以被后面的操作所看见,这两个操作的执行在时间上就存在先后顺序. 这里, 我们根据时间序的关系来对边界进行扩展, 可以大大减少下一边界的可能性, 只有在没有时间序关系时, 才需要遍历下一边界的所有的可能. 另外, 为了使得时间序的概念适用于同步一致性模型的验证, 我们还改进了时间序的概念, 通过对时间序进行放松, 保证了验证算法的正确性.

每次扩展边界时, 都需要通过在操作之间加动态边来判断这次扩展是否符合相邻边转换规则. 如果加边之后, 操作之间形成环, 则说明这次对边界的扩展不符合相邻边转换规则, 需要访问 f 的其它相邻边界; 如果没有出现环, 则以新的扩展边界 f'为参数递归调用函数 ExploreActiveFrontier. 如果 f 所有可能的下一个边界都不符合相邻边转换规则, 那么 ExploreActiveFrontier 函数返回 false. 这说明对写操作或同步块而言, 验证算法无法为它们构造出一个全序, 因此程序的运行结果不满足待验证的存储一致性模型的要求. 验证算法需要返回 false. 而如果函数最终可以到达结束边界, 则说明内存操作之间已经构造出了符合存储一致性模型的全序, 函数返回 true, 验证算法同样返回 true. 这说明程序的执行结果满足待验证的模型.

3 MODV的性能优化

MODV 工具已成功地对一种满足同步一致性模型的众核处理器进行了验证, MODV 所实现的算法的最坏时间复杂度是 $O(n^2)$, 是一种较快的验证算法. 基于软件的动态验证工具, 性能是决定工具适用性的最主要因素, 因此我们需要研究更快的验证工具. 通过对算法和程序的分析, 我们发现影响 MODV 性能的地方主要有两处. (1)建立静态边时我们使用的数据结构是邻接矩阵, 这样静态加边的时间复杂度是 $O(n^2)$. (2)另外一个影响性能的原因是我们只能间接地获得操作的时间窗口. 这样, 即使两个操作之间存在时间序, 我们也很有可能无法从操作的时间窗口中推断出该时间序. 这种情况导致了时间序关系无法得到充分利用, 增加了无效回溯的次数.

针对第一种情况, 我们的改进措施是: 采用哈希表的思想, 通过两次扫描建立静态边. 第一次扫描将具有相同地址及操作值的读写操作存储在同一位置中. 第二次扫描将同一位置内的写操作与所有读操作之间建立静态边. 这样, 就避免了使用邻接矩阵建立静态边时所有的无用检测. 由于数据的特点是同一地址上每次写入的值都不同, 因此, 同一位置上只有一个写操作. 所以, 采用这种算法, 静态加边的时间复杂度是 $O(n)$, 与原有的时间复杂度 $O(n^2)$相比, 算法的效率得到很大提升.

针对第二种情况, 我们采用了两种启发式策略: (1)如果同一个处理器中有多个写操作(写入同一地址)或者同步操作(使用同一把锁)可以作为有效操作, 那么我们可以根据程序序(program order)依次将他们选择为有效操作. 这样做的好处不仅可以避免上述问题, 而且可以保证处在临界区内的操作一定会在同步操作之后被选取, 这不但与存储一致性模型中对同步块的要求相符, 而且可以避免很多无意义的回溯. (2)加动态边时, 为同步操作添加相应的边消耗的时间最长. 由于一些同步块之间没有时间序关系, 而同步块中的操作又会对同步块的加边方式造成影响, 因此, 对同步块的加边是非常复杂的. 我们发现, 即便得不到同步块之间的时间序关系, 也可以根据同步块中操作的序关系, 提前确定同步块的顺序, 避免在算法运行时反复尝试. 确定同步块顺序的具体操作步骤是:

1) 如果写操作 w 与读操作 r 之间存在写入关系, 那么 w 所在同步块在 r 所在同步块之前.

2) 如果写操作 w 与读操作 r 之间存在写入关系, 且写操作 w' 与读操作 r 有相同地址并且有程序序关系, 那么 w' 所在同步块在 w 所在同步块之前.

3) 如果写操作 w 与读操作 r 之间存在写入关系, 且 w 与写操作 w' 有相同地址并且有程序序关系, 那么 r 所在同步块在 w' 所在同步块之前.

我们可以为同步操作额外申请一个空间, 记录与它没有时间序关系, 但是必须要在它之前执行的同步操作. 在选择下一个同步操作时, 首先检测必须要在该同步操作之前执行的同步操作是否已经加边, 只有这些操作加边之后, 该同步操作才能被选择, 通过这种方式, 可以进一步减少回溯的次数.

对第一种情况的加速措施在算法 1 的第 1 行起作用, 对第二种情况的加速措施在算法 1 的第 5 行起作用, 他们是针对验证算法的不同阶段进行的优化,

独立地发挥作用, 所以将其整合在一起并不会相互干扰. 其中针对第二种情况的两种启发式策略其实都是在得不到操作之间时间序关系时, 通过其它方式获取序关系, 只不过采用的具体方式不一样, 方法 (1) 是通过程序序, 方法 (2) 是通过读写关系, 二者是累加的关系, 并不会产生冲突. 因此, 对这些加速方法进行融合是合理的.

4 实验结果

我们使用 MODV 的基准算法[19,20]和改进算法来验证大规模程序, 通过比较 MODV 的实际运行时间, 来对其性能进行分析. 我们验证的是 Godson-T 的存储一致性模型, Godson-T 是中国科学院计算技术研究所设计并实现的一个众核处理器系统, 它采用的内存模型是一个类似域一致性模型的同步一致性模型[3]. 实验在使用了 160 核 2.4GHz 的 Intel Xeon E78870 处理器, 内存为 256GB 的 Linux 服务器上运行. 在随机生成的测试用例中, 读操作占 60%, 写操作占 30%, 同步操作占 10%. 程序中一个有 2 个地址, 1 把锁.

图 2 和图 3 分别显示了当程序运行于 8 个核和 16 个核的处理器, 操作数从 10k 增加到 100k 的情况下, 分别使用基准算法和优化后算法验证程序的运行时间. 从图中可以看出, 优化后的算法由于使用了各种加速技术, 运行时间大幅下降.

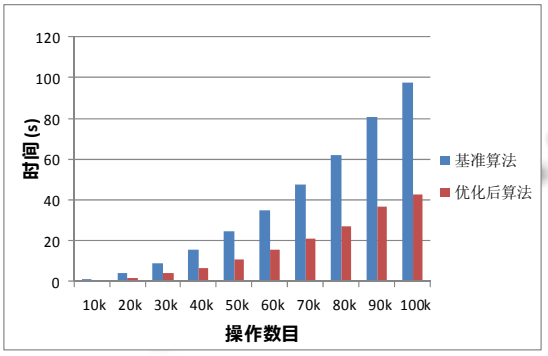

图 2 基准算法和优化算法的性能比较(8 核)

我们对 MODV 工具所使用的算法运行时间百分比进行了进一步的分析. 根据我们使用的优化方法, 针对构造静态边的优化可以确定性地将时间复杂度由 $O(n^2)$ 降低到 $O(n)$, 因而运行时间将大幅度减少. 而针对时间序的两种启发式策略要在数据比较合适时才能发挥作用, 所以与前一种优化方法比, 优势不明显. 我们对 8 核 100k 的数据量下工具的性能进行了进一步分析, 统计了三个关键函数耗时百分比, 如表 1 所示.

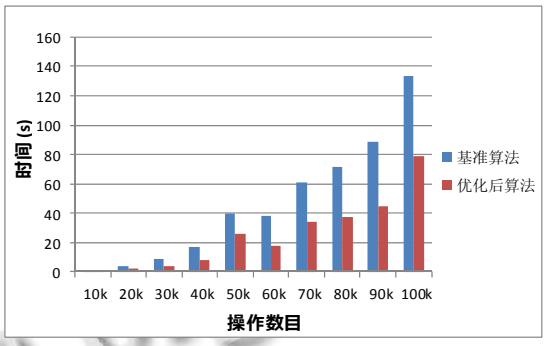

图 3 基准算法和优化算法的性能比较(16 核)

表 1 优化前后关键函数占用时间百分比

函数 / 类别 用时比(%)	construct_map	check_cycle	exploreActive_frontier
优化前	63.43	28.70	7.71
优化后	0.10	78.41	20.87

在表 1 中, construct_map, check_cycle, exploreActive_frontier 分别对应算法 1 中的 1,2,5 行, 是验证工具的三个关键函数. 从表 1 中可以看出, 用于构造静态边的函数 construct_map 在优化后, 所占用的时间百分比大幅度下降, 与我们之前的分析一致.

5 结论

同步一致性模型是一类弱存储一致性模型, 对该类模型的验证非常困难. MODV 是一种动态存储一致性模型验证工具, 可以较为有效地验证该类存储一致性模型. 但是 MODV 在性能方面仍存在不足, 无法验证更大规模的并发访存程序. 我们对 MODV 的算法和实现进行了分析, 提出并实现了多种性能加速的策略. 实验结果证明了我们的这些策略可以有效提高 MODV 在验证大规模并发程序时的性能.

参考文献

1 Adve SV, Gharachorloo K. Shared memory consistency models: a tutorial. IEEE Computer, 1996, 29 (12): 66–76.

2 Lamport L. How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. on Computers, 1979, 28(9): 690–691.

3 Fan DR, Zhang H, Wang D, Ye XC, Song FL, Li GJ, Sun NH.

Godson-t: An efficient many-core processor exploring thread-level parallelism. IEEE Micro, 2012, 32(2):38–47.

4 Hansson A, Goossens K, Bekooij M, Huisken J. Compsoc: A template for composable and predictable multi-processor system on chips. ACM Trans. on Design Automation of Electronic Systems, 2009, 14(1): 21–24.

5 Gibbons PB, Korach E. On testing cache-coherent shared memories. Proc. of the 6th Annual ACM Symposium on Parallel Algorithms and Architectures. New York: ACM. 1994. 177–188.

6 Park S, Dill DL. An executable specification, analyzer and verifier for RMO (relaxed memory order). Proc. of the 7th Annual ACM Symposium on Parallel Algorithms and Architectures. New York: ACM. 1995. 34–41.

7 Condon AE, Hill MD, Plakal M, Sorin DJ. Using Lamport clocks to reason about relaxed memory models. Fifth International Symposium on High-Performance Computer Architecture. California: IEEE COMPUTER SOC. 1999. 270–278.

8 DeOrio A, Wagner I, Bertacco V. Dacota: Post-silicon validation of the memory subsystem in multi-core designs. Fifteenth International Symposium on High-Performance Computer Architecture. California: IEEE COMPUTER SOC. 2009. 405–416.

9 Hangal S, Vahia D, Manovit C, Lu JYJ. TSOtool: a program for verifying memory systems using the memory consistency model. 31st Annual International Symposium on Computer Architecture, Proc. California. IEEE COMPUTER SOC. 2004. 114–123.

10 Manovit C, Hangal S. Efficient algorithms for verifying memory consistency. Proc. of the Seventeenth Annual ACM Symposium on Parallelism in algorithms and Architectures. New York. ACM. 2005. 245–252.

11 Roy A, Zeisset S, Fleckenstein CJ, Huang JC. Fast and generalized polynomial time memory consistency verification. 18th International Conference on Computer Aided Verification. Berlin. SPRINGER -VERLAG. 2006, 4144. 503–516.

12 Manovit C, Hangal S. Completely verifying memory consistency of test program executions. 12th International Symposium on High-Performance Computer Architectur. California. IEEE Computer SOC. 2006. 166–175.

13 Chen YJ, Lv Y, Hu WW, Chen TS, Shen HH, Wang PY, Pan H. Fast complete memory consistency verification. Fifteenth International Symposium on High-Performance Computer Architecture. California. IEEE COMPUTER SOC. 2009. 381–392.

14 Hu WW, Chen YJ, Chen TS, Cheng Q, Li L. Linear time memory consistency verification. IEEE Trans. on Computers, 2012, 61(4): 502–516.

15 Burnim J, Sen K, Stergiou C. Sound and complete monitoring of sequential consistency for relaxed memory models. Tools and Algorithms for Construction and Analysis of Systems, 2011, 6605: 11–25.

16 Chen YJ, Li L, Chen TS, Li L, Wang L, Feng XW, Hu WW. Program regularization in memory consistency verification. IEEE Trans. on Parallel and Distributed Systems, 2012, 23(11): 2163–2174.

17 Alglave J, Kroening D, Nimal V, Tautschnig M. Software verification for weak memory via program transformation. Programming Languages and Systems, 2013, 7792: 512–532.

18 Sorensen T, Gopalakrishnan G, Grover V. Towards shared memory consistency models for GPUs. Proc. of the 27th International ACM Conference on Supercomputing. New York. ACM. 2013. 489–490.

19 Lv Y, Sun LM, Ye XC, Fan DR, Wu P. Efficiently and completely verifying synchronized consistency models. Automated Technology for Verification and Analysis. Berlin. SPRINGER-VERLAG. 2014, 8837. 264–280.

20 孙鲁明.同步一致性模型的动态验证[硕士学位论文].北京:中国科学院大学,2014.

21 Lamport L. Time, clocks, and the ordering of events in a distributed system. Communications of the ACM, 1978, 21(7): 558–565.