RCC 高速缓存一致性协议的带参验证①

孙鲁明[1,2], 周 琰[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 100490)
[3](微软中国有限公司, 苏州 215000)

摘 要: Godson-T 众核处理器的 RCC 高速缓存一致性协议是一种非常有特色的带参并发系统, 对此协议的带参验证是一个很大的挑战. Cubicle 是最近出现的基于 SMT 求解器的带参模型检测工具. 我们使用了 Cubicle 带参模型检测工具, 成功对 RCC 协议进行了建模和验证. 实验结果表明, RCC 协议在结点个数为任意规模时均满足协议的各种安全性质.

关键词: 众核处理器; 缓存一致性协议; 带参模型检测; RCC; Godson-T

Parameterized Verifation of RCC Cache Coherence Protocol

SUN Lu-Ming[1,2], ZHOU Yan[3]

[1](Institute of Software, Chinese Academy of Sciences, Beijing 100190, China)
[2](University of Chinese Academy of Sciences, Beijing 100190, China)
[3](Microsoft(China) Corporation, Ltd., Suzhou 215000, China)

Abstract: RCC cache coherence protocol in Godson-T many-core processor is a characteristic parameterized concurrent system. It is a challenge to verify this protocol. Cubicle is a recently built parameterized model checking tool based on SMT solver. We used Cubicle to model and verify RCC protocol successfully. The experimental results show that RCC protocol satisfies all kinds of safety properties regardless of how many nodes it has.

Key words: many-core processor; cache coherence protocol; parameterized model checking; RCC; Godson-T

1 引言

在计算机科学的许多重要应用领域中都存在着带参并发系统. 例如 Cache 一致性协议, 安全协议, 通信协议和网络协议等. 一个规模为 N 的带参并发系统 P(N) 通常由零或者少数几个小的异构进程和 N 个同构进程组成. 由于带参系统在理论和工程中的重要性, 验证带参并发系统的正确性成为计算机学术界和工业界共同关心的课题. 验证带参系统的困难之处在于: 通常我们可以对带参系统的若干个很小的实例进行有效的验证或者测试, 但这并不能保证系统在任意规模下仍然是正确的. 我们必须找出有效的方法来验证任意规模的的带参系统的正确性.

高速缓存(Cache)一致性协议是一种典型的带参并发系统, 它是弥补多处理器计算机系统中处理器和存储器速度差距的有效方法. 为了获得更好的性能和可扩展性, Cache 一致性协议也成为形式化方法中最早的工业应用之一. Godson-T 是中国科学院计算技术研究所设计并实现的一个面向较大规模的并行处理的众核体系结构[10]. 目前 Godson-T 支持 64 个计算结点(Core)和一个专门用于同步管理的中央结点(Sync Manager). 其中的 RCC(Region-based Cache Coherence)高速缓存一致性协议(简称 RCC 协议)是 Godson-T 众核处理器所使用的缓存一致性协议, 该协议在实现中最多支持 64 个结点, 在理论上需要支持任意多个结点, 因此是一个典型的带参并发系统. 为了提高系统的并发处理能力, Godson-T 选用了一种相对复杂的存储一致性模

① 收稿时间:2014-03-11;收到修改稿时间:2014-04-14

型—Scope 一致性模型. 相应的, RCC 协议也和传统的缓存一致性协议不同, 是一种非常有特色的弱一致性协议, 因此, 该协议的带参验证是一个很大的挑战.

Cubicle 是一个基于 SMT 求解器的模型检测工具, 由 Sylvain Conchon 等人开发[6]. Cubicle 可以验证一大类基于数组的无穷状态的不变式(安全性质). 它是一个完备的符号模型检测工具, 其处理的无穷状态系统中的变量都是数组, 并通过对非安全状态的反向可达性分析来验证所要求的性质. Cubicle 可以用于验证命令行程序, 带参系统, 带时间的系统和分布式系统. Cubicle 的系统描述语言的表达能力很强, 非常适合 Cache 一致性协议的带参验证. 此外, 和其他带参模型检测工具相比, Cubicle 的验证效率是最高的. 因此我们使用 Cubicle 作为验证工具来验证 Godson-T 众核处理器的 RCC 高速缓存一致性协议.

2 相关工作

Cache 一致性协议主要分为监听协议(snoopy bus protocol)和基于目录的协议(directory-based protocol)两类. 相对来说, 基于目录的协议的验证更加困难一些, 其中 GERMAN 协议和 FLASH 协议是形式验证领域中两个作为基准协议的基于目录的协议. 模型检测方法是目前最为成功的形式验证方法. 目前通用的模型检测工具有 Candence SMV[13], NuSMV[5], Murφ[8], SPIN[12]等. 这些模型检测工具可以验证任何有穷状态系统, 但是不支持对带参并发系统的验证. 我们曾用 Murφ 对 RCC 协议进行了建模和验证, 但是最多只能验证规模为 3 个结点的协议[1]. 为了有效验证带参并发系统, 学术界提出了多种带参模型检测方法并开发了多个带参模型检测工具.

在 Cache 一致性协议的带参模型检测方法方面, 目前已有多种方法. Emerson 等人提出了基于 cutoff 的模型检测方法[9]. 这种方法充分利用了带参并发系统所具有的的对称性, 确定出参数系统 N 的一个固定的阈值(cutoff)C, 使得只要对 N 小于 C 的所有系统实例都满足所要验证的性质 F, 就可以保证系统对参数 N 的任意取值都满足 F. 也就是说, 把对原来任意大小规模的模型检测, 转化到仅仅针对有限多个小规模系统的检测来完成, 由此给出了一个确定的判定过程. 这个方法被成功用于 Cache 一致性的验证. Penuli 等人提出的隐不变量(Invisible Invariants)方法也是一种基于 cutoff 的方法[15]. 他们首先对带参系统的一个有限的实例进行模型检测计算出辅助的不变量. 然后用类似归纳不变量检测的方法自动验证系统的性质. 在这种方法中 GERMAN 协议的阈值降为 4. Chou, Mannava 和 Park 等人提出了基于参数抽象和卫士加强(parameter abstraction and guard strengthening)的方法[4]. 现在常被成为 CMP 方法. 该方法所构造的抽象模型包括所有选取的代表进程, 以及另外一个代表对所有未选进程抽象的附加进程. 该抽象模型的构造过程遵循一种反例引导逐步精化的模式, 当抽象模型不满足所验证的性质时, 需要人工分析反例, 找出一个适当的不变量公式去加强某些规则的卫士以约束该模型. 重复这个过程直至原验证性质以及所有新引进的不变量公式在抽象模型中能够被满足. Talupur 等人利用 CMP 方法成功验证了 Intel 公司的一个众核处理器的 Cache 一致性协议[14]. 谓词抽象(predicate abstraction)是一种常用的带参验证方法[11]. 通常的抽象要求用户给出一定的抽象映射关系, 从而计算出具体系统所对应的的抽象系统的形式. 谓词抽象可以减轻用户的负担, 它仅仅需要由人工所给出一个与系统的性质相关的一些小的由谓词组成的集合(假定集合包含 k 个元素), 它们描述了系统状态上可能的某些性质. 利用这些谓词, 可以计算出一个有穷抽象模型, 至多包含 2^k 个状态. 通过计算抽象模型的可达状态空间, 可以生成一个以所提供的谓词所能表达的最强的不变量. 通过抽象映射关系的逆映射, 计算出这个不变量所对应的的具体形式则可以证明原来具体系统的某些性质. 谓词发现技术则试图自动分析系统的特性, 以经验型的启发式算法自动找到有用的谓词集合. 谓词发现和谓词抽象技术的结合, 为非有穷系统的安全性质给出了一个全自动化的验证过程. 应用这种方法, Das 等人分析了 FLASH 协议的简化版本[7].

近年来在学术界和工业界已存在多种直接验证带参系统的模型检测工具, 可以用于验证 Cache 一致性协议. 除了 Cubicle 之外, 典型的带参模型检测工具还有 PFS[3], Undip[2]等. PFS 是 UPPSALA 大学开发的一个带参系统的验证工具. PFS 主要通过对系统过近似的可达状态进行反向搜索的原理来验证系统的安全性质. 在 PFS 中, 一个结点被看做一个有若干局部变量(在有穷域上取值)的有穷自动机. 自动机的迁移是由结点的状态, 局部变量和全局变量所决定. 结点可以

在系统运行时动态地添加和删除. PFS 主要的思想就是把一个迁移关系看成包含在带参系统中过近似的关系. 为了做到这一点, 他们通过排除不满足条件的结点的方式对全称量词的语义进行修改. 近似的迁移系统对整个系统来说是一个单调的关系. 事实上全称量词是唯一的不单调的操作, 因此在模型中是唯一会被近似的地方. 正是由于近似系统是单调的, 能够使用基于符号的反响可达状态搜索算法来分析. PFS 成功验证了很多简单协议, 但是无法验证像FLASH协议这种复杂的协议. Undip 也是 UPPSALA 大学开发的一个无上界分布式带参系统验证工具. 它可以验证分布式或者非分布式的带有无上界变量的带参系统的安全性. Undip 也是基于通过对系统过近似的可达状态进行反向搜索的原理. 在 Undip 中差值有界矩阵被用来定义和操作对于整数变量的约束.

3 RCC协议描述

对于一个共享内存的多核处理器系统, 其存储一致性模型定义了它的访存操作所满足的约束条件, 即读写操作的规则以及读写操作如何访问内存. 缓存一致性协议是保证存储一致性模型的重要组成部分. 缓存一致性协议主要是为了确保一个共享内存的多核系统的多副本数据间的一致性. 缓存一致性协议一般来说对用户是透明的. 要做到这一点, 缓存一致性有两个约束: 单写多读约束和数据值约束. 满足这两个性质的缓存一致性是一种强的一致性协议. 在基于锁的弱一致性的内存模型中, 为了能够使强一致性的程序正确运行, 系统提供了一系列临界区(也叫做锁)的指令, 来限制弱一致性下的乱序执行. 所谓临界区, 就是多个进程必须互斥地对它进行访问. 临界区一般用锁来实现, 当一个进程获取锁 L 后, 其他进程想要再获取锁 L, 必须等之前的进程执行完毕, 并释放掉锁 L 后才能取得. Godson-T 处理器采用了弱一致性模型中的 Scope 一致性. 为了适应 Godson-T 的弱一致性模型, RCC(Region Cache Coherence)协议是一个和一般的缓存一致性协议不同的弱一致性协议. 它的锁管理器相关的部分是对用户可见的. 在 RCC 协议中, 有一个全局的锁管理器来统一对结点的锁进行分配和管理. 每个结点可以通过 Acquire 和 Release 操作来获取和释放所. 每一把锁都有一个 ID 来标记, 每个结点获取到某个 ID 的锁后, 读写操作就能通过使用该 ID 的锁进行. 同时, 同一个 ID 在某个时刻至多只能被一个结点占用. 带锁的操作也称为临界区内的操作, 不带锁的操作也被称作临界区外的操作.

RCC 协议的非形式化化描述如下[1]:

临界区外的读操作(不带锁的读) 判断 Cache 中是否有该副本, 若有, 则直接读取; 若无, 对 Cache 当前副本执行 Replace 操作, 再从 Memory 中读取, 写入 Cache 中.

临界区外的写操作(不带锁的写) 采用写回策略, 判断 Cache 中是否有该副本, 若有, 则将新值写入 Cache 中, 修改 Cache 状态为 Dirty; 若无, 对 Cache 当前副本执行 Replace 操作, 再将新值写入 Cache 中, 修改 Cache 状态为 Dirty.

临界区内的读操作(带锁的读) 判断是否为该锁的首次读, 若是, 对 Cache 当前的副本执行 Replace 操作, 再从 Memory 中读取, 并写入 Cache; 若否, 读过程类似于临界区外的读操作.

临界区内的写操作(带锁的写) 采用写穿透策略, 首先将新值直接写入 Memory, 其次判断 Cache 中是否有该副本, 若有, 再用新值更新 Cache 副本.

替换操作(Replace) 当 Cache 中的副本状态为 Dirty, 并且需要被替换出 Cache 时, 会将当前副本的值写入 Memory.

从 RCC 协议的描述中可以看出, RCC 协议的性质与是否在临界区内紧密相关, 因此必须在性质描述中显示地给出临界区的相关概念. RCC 协议如果是在临界区内的读写操作, 那么就是强一致性, 根据其描述可以看出, 临界区内的读写操作的行为与顺序一致性相同; 而如果是非临界区的读写操作, 那么没有任何一致性保证, 仅仅是在缓存需要替换操作的时候才会对内存进行操作, 也就是说结点与结点之间临界区外的操作时可以任意乱序执行的. 这种多粒度的一致性是 RCC 协议的一个特色: 强一致性下效率低, 访存频率高, 数据一致性高; 弱一致性下效率高, 访存频率低, 数据一致性低. 程序员可以根据程序要求的一致性来选择是否需要临界区.

4 Cubicle带参模型检测工具

Cubicle 是一个基于 SMT 求解器的模型检测工具, 由 Sylvania Conchon 等人开发[6]. Cubicle 可以验证一大类基于数组的无穷状态系统的安全性质. 它是一个完

备的符号模型检测工具, 其处理的无穷状态系统中的变量都是数组, 并通过对非无穷状态的反向可达性分析来验证所要求的性质. Cubicle 中的状态变量和迁移关系均可以表示为受限的一阶逻辑公式, 因此, 反向可达性分析被转化成受限的一阶逻辑公式的 SMT 求解过程.

Cubicle 的系统描述语言和 Murφ 类似, 增加了对带参系统的表示. 下面是 Cubicle 系统描述语言的简介. 在 Cubicle 中, 一个系统的系统状态变量分为普通变量和数组变量, 一个逻辑公式表示初始状态, 状态的变化由状态迁移集合表示. 系统的参数由一个专用类型 proc 进程标识符来表示系统中存在任意多个进程/结点. 此外, Cubicle 还提供了标准的数据类型如 int, real 和 bool 等. 系统的状态由全局状态变量和 proc 为索引的数组变量的状态构成. 系统的迁移采用通常的卫士/动作方式, 迁移可以用进程标识符表示迁移可以扩展到任意多个进程. 系统的执行满足 UNITY 语言: 整个系统的执行是一个无穷循环, 每步执行均在满足卫士条件的所有迁移实例中非确定性地选择一个迁移实例执行; 根据该迁移实例的动作方式更新系统的状态.

5 RCC协议的建模和验证

RCC 协议的结点个数是任意的, 因此我们需要考虑协议的带参建模. 这个主要体现在结点的数据结构和结点的状态迁移方面. 由于 Godson-T 协议的缓存协议是弱一致性的, 在建模的时候必须将内存和缓存放在一起建模. 我们将建模粒度选取为获取和释放锁, 读写操作这一层面, 将缓存块的选择和替换以过程的形式隐含在每个操作中. 建模过程如下:

首先是用户定义的类型, 这里定义了 Cache 块的三种状态.

```
(*user-defined type*)
    type cacheState = Invalid | Dirty | Valid
```

系统除了协议的结点个数是任意的, 其他变量都是有限个, 因此我们需要先确定待验证系统的规模, 如下所示.

```
(*Lock set*)
    var Lock_1_owner : proc
    var Lock_1_beUsed : bool
    var Lock_2_owner : proc
    var Lock_2_beUsed : bool
(*Memory*)
    var Memory_1_data : int
    var Memory_2_data : int
    var Memory_3_data : int
```

在这个实例中我们确定了系统有两把锁和三个内存地址. 在确定了系统规模之后, 我们确定系统的全局变量, 如下所示.

```
(*Node*)
    array Cache_1_state[proc] : cache
    array Cache_1_addr[proc] : int
    array Cache_1_data[proc] : int
    array Cache_2_state[proc] : cache
    array Cache_2_addr[proc] : int
    array Cache_2_data[proc] : int
    array FirstRead_a1[proc] : bool
    array FirstRead_a2[proc] : bool
    array FirstRead_a3[proc] : bool
    array HasLock[proc] : bool
```

协议中存在任意多个结点 node, 因此所有的数据都是 proc 数组类型, 表示结点的个数是任意的. 一个 node 数据表示一个执行结点主要包括: 缓存(cache), 临界区标志和首次读标志. 其中 cache 由三个数组构成, 数组分别表示 cache 单元的状态, 地址和数据. 临界区标志 HasLock 和首次读标志 FirstRead 用于协议的状态控制.

系统的初始状态用一个一阶逻辑公式表示, 这里 z 是进程标识符.

```
(*Init*)
init(z) {
    Lock_1_beUsed = False &&
    Lock_2_beUsed = False &&
    Memory_1_data = 0 &&
    Memory_2_data = 0 &&
    Memory_3_data = 0 &&
    Cache_1_state[z] = Invalid &&
    Cache_2_state[z] = Invalid &&
    FirstRead_a1[z] = True &&
    FirstRead_a2[z] = True &&
    FirstRead_a3[z] = True &&
    HasLock[z] = False
}
```

接下来我们定义协议的各种迁移, 迁移的表示也

是带参的. 首先以获取锁为例, 说明迁移的建模过程. 获取的锁的迁移如果成功执行, 首先需要检测其 guard 是否满足, guard 是一个 bool 公式. 如果 guard 为 true, 执行 action, 进行一些赋值操作. 下面是一组获取锁的迁移, n 是迁移的参数.

```
transition 11_Acquire(n)
requires { HasLock[n] = False && Lock_1_beUsed = False && Lock_2_beUsed = False }
{
    Lock_1_beUsed := True;
    Lock_1_owner := n;
    FirstRead_a1[j] := case
    | j = n : True
    | _ : FirstRead_a1[j];
    FirstRead_a2[j] := case
    | j = n : True
    | _ : FirstRead_a2[j];
    FirstRead_a3[j] := case
    | j = n : True
    | _ : FirstRead_a3[j];
    HasLock[j] := case
    | j = n : True
    | _ : HasLock[j];
}
```

在这组迁移的任意一个实例中, requires { HasLock[n] = False && Lock_1_beUsed = False && Lock_2_beUsed = False }是该迁移的卫士条件, 后面是迁移的动作. 只有该条件为真时, 迁移的动作才能发生.

我们对读写操作进行了拆分, 按照读写操作的类型分为如下多个迁移规则: 1).临界区外, 缓存中没有副本的读操作; 2).临界区内, 首次读操作; 3).临界区内, 非首次, 缓存中没有副本的读操作; 4).临界区外, 缓存中没有副本的写操作; 5).临界区外, 缓存中有副本的写操作; 6).临界区内, 缓存中没有副本的写操作; 7).临界区内, 缓存中有副本的写操作. 我们分别对这些操作进行构造迁移规则. 下面是其中一个操作的实例.

```
(*Transition a[1-3]C[1-2]A[1-3]_Lock[1-2] FirstRead A[1-3] Dirty Cache Replace *)
requires { HasLock[n] = True && Lock_1_beUsed = True && Lock_1_owner = n && FirstRead_a1[n] = True && Cache_1_state[n] = Dirty && Cache_1_addr[n] = 2 && Cache_2_addr[n] <> 1 }
{
    Memory_2_data := Cache_1_data[n];
    Lasting_2 := False;
    Cache_1_state[j] := case
    | j = n : Valid
    | _ : Cache_1_state[j];
    Cache_1_addr[j] := case
    | j = n : 1
    | _ : Cache_1_addr[j];
    Cache_1_data[j] := case
    | j = n : Memory_1_data
    | _ : Cache_1_data[j];
    FirstRead_a1[j] := case
    | j = n : False
    | _ : FirstRead_a1[j];
}
```

最后, 我们定义 RCC 协议中的安全性质. 如下所示的一个控制部分的性质是, 如果任意两个结点都已申请到锁, 那么这两个结点申请的锁不能是同一个锁.

```
(*Invariance*)
unsafe (z1 z2) { HasLock[z1] = True && HasLock[z2] = True && Lock_1_owner = Lock_2_owner }
```

6 实验结果

我们在一台 64 位的 Linux 服务器上进行了实验. 服务器的配置是 4 路 8 核 Intel Xeon 处理器, 每个核的主频为 2.9GHz, 服务器可用内存为 384GB, 操作系统为 Linux 2.6.18, Cubicle 版本为 cubicle-0.5. 虽然验证的 RCC 协议在结点个数上是无上界的, 但是每个结点中 Cache 块的个数和系统中内存地址的个数仍然是有限的. 我们分别对不同规模和类型的模型进行了验证. 在结果中我们用 c 表示每个结点内 cache 块的个数, m 表示系统中内存地址的个数, l 表示系统中互斥锁的个数, 这里固定为 2 个. 实验结果见表 1 和表 2.

表 1　RCC 协议控制性质验证结果

控制性质	内部结点个数(个)	调用求器次数(次)	耗时	内存(MB)
l2c1m2	80	59	0.06s	307
l2c2m2	182	95	0.12s	364
l2c2m3	243	119	0.16s	365

我们分别对协议的控制部分性质和数据部分性质

进行了验证. 从实验结果中可以看出, 当 c,m 这两个参数分别增大时, 状态空间也在迅速增长. 对控制部分的性质, Cubicle 可以迅速完成验证, 然而对于复杂的数据性质, Cubicle 需要用很大的内存空间在非常长的时间才能完成验证. 从这项数据可以看出Cubicle在验证不同类型的性质时效率相差非常大.

表 2 RCC 协议数据性质验证结果

数据性质	内部结点个数(个)	调用求解器次数(次)	耗时	内存(MB)
l2c1m2	12598	1549902	28.6s	463
l2c2m2	165513	3090102	45m29s	1515
l2c2m3	707681	14764692	7h1m54s	5494

7 结论

Godson-T 的 RCC 高速缓存一致性是一种相对复杂的带参并发系统, 对 RCC 协议的带参验证具有很大的难度. 我们使用了 Cubicle 带参模型检测工具, 成功地对 RCC 协议进行了建模和验证. 实验结果表明, RCC 协议在结点为任意规模时均满足其控制和数据性质. Cubicle 尚不能对内存地址个数以及 Cache 块个数等参数进行带参验证. 此外, 对于复杂的数据性质, Cubicle 的验证效率仍有问题. 这些问题需要在将来的研究中加以解决.

参考文献

1 周琰.Godson-T 缓存一致性协议的 murphi 建模和验证.计算机系统应用,2013,22(10):124–128.

2 Abdulla PA, Delzanno G, Rezine A. Parameterized verification of infinite-state processes with global conditions. Computer Aided Verification. Springer Berlin Heidelberg, 2007: 145–157.

3 Abdulla PA, Delzanno G, Henda NB, et al. Regular model checking without transducers (on efficient verification of parameterized systems). Tools and Algorithms for the Construction and Analysis of Systems. Springer Berlin Heidelberg, 2007: 721–736.

4 Chou CT, Mannava PK, Park S. A simple method for parameterized verification of cache coherence protocols. Formal Methods in Computer-Aided Design. Springer Berlin Heidelberg, 2004: 382–398.

5 Cimatti A, Clarke E, Giunchiglia F, et al. NuSMV: A new symbolic model verifier. Computer Aided Verification. Springer Berlin Heidelberg, 1999: 495–499.

6 Conchon S, Goel A, Krstić S, et al. Cubicle: A parallel SMT-based model checker for parameterized systems. Computer Aided Verification. Springer Berlin Heidelberg, 2012: 718–24.

7 Das S, Dill D L, Park S. Experience with predicate abstraction. Computer Aided Verification. Springer Berlin Heidelberg, 1999: 160–171.

8 Dill DL. The Murφ verification system. Computer Aided Verification. Springer Berlin Heidelberg, 1996: 390–393.

9 Emerson EA, Kahlon V. Reducing model checking of the many to the few. Automated Deduction-CADE-17. Springer Berlin Heidelberg, 2000: 236–254.

10 Fan D, Zhang H, Wang D, et al. Godson-T: An efficient many-core processor exploring thread-level parallelism. Micro, IEEE, 2012, 32(2): 38–47.

11 Graf S, Saïdi H. Construction of abstract state graphs with PVS. Computer aided verification. Springer Berlin Heidelberg, 1997: 72–83.

12 Holzmann GJ. The model checker SPIN. IEEE Trans. on Software Engineering, 1997, 23(5): 279–295.

13 Cadence Berkeley Labs. Cadence smv, http://www.kenmcmil.com/smv.html, 1998.

14 O'Leary J, Talupur M, Tuttle MR. Protocol verification using flows: An industrial experience. Formal Methods in Computer-Aided Design, 2009. FMCAD 2009. IEEE, 2009: 172–179.

15 Pnueli A, Ruah S, Zuck L. Automatic deductive verification with invisible invariants. Tools and Algorithms for the Construction and Analysis of Systems. Springer Berlin Heidelberg, 2001: 82–97.