Godson-T 缓存一致性协议的 Murphi 建模和验证①

周 琰

[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)

摘 要: Godson-T 缓存一致性协议是用于 Godson-T 众核处理器的缓存一致性协议. 在 Godson-T 协议中, 缓存一致性协议和存储一致性模型存在紧密的紧耦合关系, 分析协议的一致性时发现该协议满足的缓存一致性不是强一致性, 不满足传统意义上缓存透明的一致性要求. 我们选取了 Murphi 模型检测工具作为我们建模的语言和验证工具. 在对 Godson-T 缓存一致性协议建模的时候, 由于协议的上述特点, 我们需要对处理器核结点, 高速缓存和内存作为一个整体建模, 并成功地验证了协议的相关性质.

关键词: 众核处理器; 内存一致性模型; 缓存一致性协议; 模型检测

Modeling and Verification of Godson-T Cache Coherence Protocol with Murphi Tool

ZHOU Yan

[1](Institute of Software, Chinese Academy of Sciences, Beijing 100190, China)
[2](University of Chinese Academy of Sciences, Beijing 100190, China)

Abstract: Godson-T cache coherence protocol is used in Godson-T many-core architecture. In Godson-T, there is a close tight coupling relationship between cache coherence protocol and memory consistency model. In our research of Godson-T, we found that its cache coherence is relaxed unlike other cache coherence protocols. We choose Murphi as modeling language and verification tool. Thus, when we modeling Godson-T, core, cache and memory must be take into account as a whole. Some invariants and properties has been verified with Murphi.

Key words: many-core processer; memory consistency model; cache coherence protocol; model checking

随着单核处理器越来越不能满足人们的需要, 多核与众核处理器逐渐成为了目前的主流. 如今, 无论是个人电脑还是大型服务器, 甚至是智能手机, 都在使用多核或众核处理器. 多核和众核处理器的发展中存在着一系列重大的挑战. 其中一个重要的挑战就是如何验证多核处理器及其高速缓存之间的一致性问题. 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, 其中 Godson-T 缓存一致性协议(简称 Godson-T 协议)是 Godson-T 众核处理器所使用的缓存一致性协议[1]. Godson-T 协议旨在解决处理核、缓存和内存三者之间的数据一致性问题. Godson-T 协议没有使用业界经典的内存模型和缓存一致性协议, 是一个非常有特色的协议, 因此对其建模和验证是一个挑战.

对于一个一致性协议的建模和验证有很多种方式. 一般来说, 缓存一致性协议都是强一致性的, 因此都是先将缓存一致性协议从存储系统中剥离出来, 用形式化语言进行描述和建模, 通常模型检测的方法是用状态迁移系统表示模型, 用时序逻辑公式表示待验证性质, 然后使用模型检测工具进行全自动验证. 目前有多种模型检测工具, 例如 Cadence SMV[2], TLV[3], NuSMV[4,5], Murphi[6]等. 把缓存一致性协议剥离存储系统后, 即可将缓存看成一个对系统透明的部分, 然后研究其存储一致性模型符合哪种内存模型, 常用的存储一致性模型包括顺序一致性(SC)模型, TSO 模型, 释放一致性(RC)模型等[7]. Godson-T 协议的特殊性在于它无法直接拆分成一个存储一致性模型和一个缓存

① 基金项目:国家自然科学基金(61272135)
收稿时间:2013-03-22;收到修改稿时间:2013-04-18

一致性模型进行分别研究, 因此如何将两者混合在一起建模和验证就成为我们研究的焦点.

1 描述

1.1 内存模型与缓存一致性

对于一个共享内存的多核处理器系统, 它的正确性分为存储模型的一致性(consistency)和缓存一致性(coherence). 存储模型的一致性定义了共享内存的一致性. 它包括了读写操作的规则以及读写操作如何影响内存. 而缓存一致性的定义主要是如何确保一个共享内存的多核系统的多副本数据间的一致性. 一般来说, 存储一致性模型是对程序员和用户可见的, 系统会根据自身存储模型的一致性来提供各种 FENCE 指令让程序员调用. 而缓存一致性一般来说保证了可以将缓存看成一个不可见的部分, 程序员和用户不需要考虑缓存的一致性. 要做到这一点缓存的一致性有两个约束: 单写多读(Single-Write, Multiple-Read(SWMR))约束和数据值(Data-Value)约束. SWMR 是指对于任何一块内存单元, 任意时刻最多只有一个核在写操作或者一些数量(一个或者多个)的核在进行只读操作. Data-Value 约束是指一个内存单元的值应当在任何一个操作开始时和结束时的值是一样的. 缓存一致性协议一般有基于侦听总线和基于目录两种. 基于侦听总线的缓存一致性协议主要是指核与核之间, 核与内存之间通过总线相连, 每一个核都可以看见总线上的请求. 这样在总线上就有一个对所有核和内存都可见的一致性序. 基于目录的缓存一致性协议是指在系统中有一个主(HOME)结点, 由它来负责决定和产生出一个一致性序, 每个核和内存与主结点之间是单播传递消息的[7]. 在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, 其他进程要想再获取锁 L, 必须等之前的进程执行完毕释放掉锁 L 后才能获取.

1.2 Godson-T 协议的非形式化描述

Godson-T 协议没有使用基于目录的 cache 一致性协议, 它选择了弱一致性中基于锁的域一致性作为片上的存储模型[1]. 在 Godson-T 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 操作来获取和释放锁. 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, 读写操作就通过使用该 ID 的锁进行.

另外, 同一时间, 同一个 ID 的锁至多只能被一个结占用. 带锁的操作也称作临界区内的操作, 不带锁的操作也称作临界区外的操作. Godson-T 协议描述如下所示:

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.

② 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.

③ 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 当前副本执行 Replace 操作, 再从 Memory 中读取, 写入 Cache; 若否, 读过程类似于临界区外的读操作.

④ 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.

⑤ 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, 会将当前该副本的值写入 Memory.

2 一致性分析

一般意义上存储模型规定了计算机的数据一致性问题, 是程序员可见的; 而缓存一致性协议是一个强一致性的协议, 对程序员是透明的. Godson-T 协议的特殊性在于它无法直接分离为内存模型和缓存一致性协议, 其根本原因在于它的缓存一致性协议并不是强一致性的, 无法将其看成一个对上下都透明的协议. 同时, 从 Godson-T 协议的说明中我们可以看出, 缓存的一致性的强弱与是否在临界区内有着紧密的联系, 这就使得 Godson-T 协议中的缓存和内存模型是无法分离的. 尽管无法将 Godson-T 协议的存储模型和缓存一致性协议分离, 我们仍可以对 Godson-T 协议的一致性进行分析. 我们主要通过具体的例子来比较 Godson-T 协议与主流内存模型的一致性上的区别, 这里我们假设主流内存模型的缓存一致性都是强一致性的.

2.1 Godson-T 协议一致性与释放一致性

释放一致性是 Gharachorloo 等人提出的一种弱一致性模型[7]. 它提供了 Acquire 和 Release 操作来实现类似 Fence 操作的功能. 与 Fence 不同的是 Acquire 和 Release 只保证单向内存访问顺序, 如下所示:

$\text{Acquire} \rightarrow \text{Load,Store}(\text{no Load,Store} \rightarrow \text{Acquire})$

$\text{Load,Store} \rightarrow \text{Release}(\text{no Release} \rightarrow \text{Load,Store})$

而 Godson-T 协议中的 Acquire 和 Release 不提供上述保证. 因此 Godson-T 协议的一致性要比释放一致性要弱. 下面用一个例子来说明它们之间的强弱,见图 1 所示.

P1	P2
X=1; Acquire(L); Y=2; Release(L);	Acquire(L); A=Y; Release(L); B=X;

图 1　程序例子

我们假设 P1 比 P2 先获得锁 L, 那么根据上述释放一致性的描述, P2 中语句 B=X 一定能得到 P1 中写入的 X 的值, 即 B = 1. 但是, 在 Godson-T 协议中, X=1 是属于临界区外的写操作, 因此新值的传播没有任何保证, 导致了 P2 中 B 的值可能是旧值也可能是 1.

2.2 实际中 Godson-T 协议的一致性

在实际使用中的 Godson-T 协议如果是在临界区内的读写操作, 那么就是强一致性的, 根据其描述可以看出, 临界区内的读写操作与一致性最强的顺序一致性是一样的; 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可以任意乱序执行. 这样一强一弱的一致性是很实用的: 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.

3　Godson-T协议的Murphi建模

模型检测是一种重要的验证技术, 它最大的优点在于自动化, 不需要人工干预. 我们将用模型检测技术对 Godson-T 协议的重要性质进行验证. 在描述系统的语言中, Murphi 语言相比于其他语言更加接近高级语言, 并且功能强大, 简明扼要, 容易阅读. 考虑到 Godson-T 协议的复杂性, 我们选择了相对来说比较强大的 Murphi 语言来建模.

3.1 Murphi 概述

Murphi 是由斯坦福大学 David Dill 教授领导的科研小组开发的. Murphi 是一个模型检测工具, 并且有自己的输入语言 Murphi 语言. 由于 Murphi 的广泛运用以及 Murphi 语言描述的系统比较简明易读, 并且它是以 guard → action 的形式书写的. Murphi 语言支持的简单数据类型有: 有界区域、枚举类型、数组和复合数据(record)对象, 高级的数据有标量集合(scalarset). Murphi 有一个基于状态集合枚举的验证器. 枚举到的状态将保存在一个哈希表中. Murphi 编译器把 Murphi 语言转化为 C++语言, 然后在对 C++语言的源文件进行编译执行. Murphi 工具的语义是 UNITY 语义, 非常适合对缓存一致性协议进行建模和验证, 被广泛应用于各种硬件系统的验证[6].

一个 Murphi 程序由下面三部分组成: 1)常量、类型和变量定义; 2)过程和函数的定义; 3)迁移规则, 初始状态和不变式. 一个 Murphi 程序对应一个状态转换图. 一个状态就是所有变量的一个赋值. 定义好初始状态以后, 状态之间的迁移由 rule 定义. Murphi 语言和 C++非常接近, 它支持定义过程和函数, 支持各种循环控制语句, 如 if-else、while、for、switch 等. Murphi 语言另一个强大之处在于它支持 ruleset 的定义, 它可以将一组迁移写成一个迁移集合, 让迁移中的变量以参数的形式书写. 一个 ruleset 可以自动根据参数值的每一种取值生成出对应的 rule. Murphi 支持带参的表示方式, 使得它非常适合对 Godson-T 协议这种带参并发协议进行建模.

3.2 Godson-T 协议的 Murphi 建模

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:

首先我们使用一些参数变量定义系统规模, 如表 1 所示.

表 1 系统规模变量表

NUM_NODE&	结点个数
NUM_LOCK	锁的个数
NUM_CACHE	结点内缓存个数
NUM_DATA	数据可表示范围
NUM_ADDR	地址范围

定义好系统规模后, 我们建模系统数据结构, 如表 2 所示.

表 2 系统数据结构表

memory	表示内存的数据结构
lock	表示锁管理器的数据结构
node	表示结点的数据结构
random	缓存替换时用到的随机变量

我们以 node 数据结构为例, 说明建模过程. 一个 node 数据表示一个执行结点, 主要包含: 缓存(cache)、临界区标志和首次读标志. 其中 cache 又是一个数据结构, 一个 cache 单元包含状态、地址(内存)和数据. 临界区标志和首次读标志是系统控制用标志. 如表 3 所示.

表 3 node 数据结构

```
CACHE : record
   state : CACHE_STATE;
   addr : TYPE_ADDR;
   data : TYPE_DATA;
end;
NODE : record
   Cache : array [TYPE_CACHE] of CACHE;
   hasLock : FLAG_LOCK;
   firstRead : array [TYPE_ADDR] of FLAG_FIRSTREAD;
end;
```

缓存的选择和替换不是一个迁移, 而是隐含在每个迁移过程中, 因此我们在 Murphi 中将其定义为一个 function 和一个 procedure, 如表 4 所示.

接下来我们将获取和释放锁, 读写操作看成原子操作, 用 ruleset 来建模. 当这些操作发生时, 系统发生相应的迁移. 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, 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. 如表 5 所示.

对于读写操作, 我们细分为下面几个原子操作: 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. 我们将这些原子操作分别用 ruleset 来建模.

表 4 缓存的选择和替换建模

```
   function getLoc(i:TYPE_NODE) : TYPE_CACHE;
   begin
  for j:TYPE_CACHE do
  if node[i].cache[j].state = INVALID then
   return j;
   endif;
  endfor;
  return random;
   end;
   procedure replace(i:TYPE_NODE; j:TYPE_CACHE);
   begin
  if node[i].cache[j].state = DIRTY then
  node[i].cache[j].state := INVALID;
  memory[node[i].cache[j].addr].data := node[i].cache[j].data;
  endif;
   end;
```

表 5 获取锁的 ruleset

```
   ruleset i:TYPE_NODE; l:TYPE_LOCK do
  rule "Acquire"
  !node[i].hasLock & !lock[l].beUsed
  ==>
  begin
   lock[l].beUsed := true;
   lock[l].owner := i;
   node[i].hasLock := true;
   for j:TYPE_ADDR do
      node[i].firstRead[j] := true;
   endfor;
   end;
```

最后, 我们将定义 Godson-T 协议中的性质, 在这里我们选择了锁不能嵌套的性质, 如表 6 所示.

表 6 系统性质描述

```
   ruleset i:TYPE_NODE do
  invariant "one node one lock restrict"
  node[i].hasLock -> ((exists j:TYPE_LOCK do lock[j].beUsed &
lock[j].owner = i endexists) & !(exists m:TYPE_LOCK; n:TYPE_LOCK
do m != n & lock[m].beUsed & lock[n].beUsed & lock[m].owner = i &
lock[n].owner = i endexists))
   end;
```

4 实验结果

我们在一台 X86-64 架构的 Linux 服务器上进行了实验. 服务器的配置是 4 路 8 核 Inter Xeon 处理器, 每个核为 2.9GHz, 300GB 内存, 操作系统为 Linux 2.6.18, Murphi 版本为 cmurphi5.4.4. 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 n 表示结点个数, c 表示每个结点内 cache 的个数, m 表示系统 memory 的个数, 带*表示屏蔽了临界区外的操作, -表示在 300GB 的内存空间内无法完成验证, 实验结果见表 7.

表 7 实验结果数据

	状态数	迁移数	耗时（秒）	内存（MB）
n2c1m2*	1890	9942	0.14	0.08
n2c1m3*	18882	126022	0.30	0.60
n3c1m2*	41570	240008	0.47	1.79
n3c1m3*	928706	6670376	16.79	29.72
n2c2m3*	908100	6837296	20.67	39.05
n3c2m3*	-	-	-	-
n2c1m2	72786	1049698	1.55	3.13
n2c1m3	1021514	20852118	67.76	32.69
n3c1m2	6008858	125655698	1380.21	258.38
n3c2m3	-	-	-	-

我们在运行以上实例的时候, 选择了 Murphi 的状态压缩和对称规约选项, 尽可能地缓解状态空间爆炸问题. 尽管如此, 我们发现当 n, c, m 这三个参数分别增大的时候, 状态空间也在迅速增长. 从实验结果可以发现, 屏蔽临近区外操作的实例都要比完整的实例的状态要少, 而且随着上述三个参数的增加, 它们之间的差距也逐渐增加. 而且, 屏蔽临界区外操作的实例的迁移数要比完整实例少很多, 这说明了系统大部分迁移是临界区外操作. 通过分别观察 n, c, m 这三个参数对系统的影响我们发现, c 的值从 1 变到 2 会使系统的状态数和迁移数显著增加, 那是因为当 c 的值为 1 的时候, 不存在 cache 的选择问题; 当 c 的值为 2 的时候, 在需要替换 cache 时需要先进行选择. 同时, n 值增加对系统状态空间的影响要比 m 值增加对系统状态空间的影响要大. 无论是否屏蔽临界区外操作, 当 n=3, c=2, m=3 的时候, 系统的状态空间已经十分巨大, 在 300GB 内存空间内 Murphi 已经无法完成协议的验证.

5 结论和下一步工作

Godson-T 缓存一致性协议是 Gonson-T 众核处理器所使用的缓存一致性协议. Godson-T 协议没有使用业界经典的内存模型和缓存一致性协议, 是一个非常有特色的协议, 因此对其建模和验证是一个挑战. 我们根据其内存模型和缓存一致性协议的强耦合特性性, 对内存和缓存同时进行建模和分析. 我们使用 Murphi 语言对 Godson-T 协议的模型进行描述, 并且使用 Murphi 模型检测工具对其进行验证.

我们发现当规模越来越大时, 状态空间爆炸问题就凸显出来, 验证所需的时间和空间均在快速增长. 当系统规模达到一定程度时, 已经无法用 Murphi 继续进行验证. 这时候, 需要研究新的验证办法. 下一步, 我们将继续精化 Godson-T 协议的建模, 使模型能够更加符合实际系统. 同时, Godson-T 是一个典型的带参系统, 通过上面的验证也只是验证协议在有限规模下的正确性, 没有验证任意规模下的正确性. 在下一步的工作中, 我们准备使用带参模型检测方法来验证所有规模下模型的正确性.

参考文献

1 林伟,叶笑春,宋风龙,张浩.众核处理器中使用写掩码实现混合写回/写穿透策略.计算机学报,2008,31(11).

2 Cadence Berkeley Lab. Cadence SMV.http://www.kenmcmil.com/smv.html, 1998.

3 Shahar E, Pnueli A. The TLV system and its applications [Master Thesis]. Weizmann Institute of Science. Israel, 1996.

4 Cimatti A, Clarke E, Giunchiglia F, Roveri M. NuSMV: A new symbolic model verifier. In: Halbwachs N, Peled D, eds. CAV 1999. LNCS, Springer Berlin Heidelberg, 1999, 1633: 682–682.

5 Cimatti A, Clarke E, Giunchiglia E, Giunchiglia F, Pistore M, Roveri M, Sebastiani R, Tacchella A. NuSMV 2: An opensource tool for symbolic model checking. In: Brinksma E, Larsen K, eds. CAV 2002, LNCS, Springer Berlin Heidelberg, 2002, 2404: 359–364.

6 School of Computing. Stanford: Murphi Model Check. http://www.cs.utah.edu/formal_verification/Murphi/.

7 Sorin DJ, Hill MD, Wood DA. A primer on memory consistency and cache coherenc. Synthesis Lectures on Computer Architecture, 2011.