迁移系统规范的并行实现正确性证明

Frank S. de Boer[1], Einar Broch Johnsen[2], Violet Ka I Pun[3], 和 S. Lizeth Tapia Tarifa[2]

[1] CWI，荷兰 F.S.de.Boer@cwi.nl[2] 挪威奥斯陆大学信息学系，挪威
{einarj,sltarifa}@ifi.uio.no[3] 西挪威应用科学大学计算系，挪威
Violet.Ka.I.Pun@hvl.no

arXiv:2302.04661v1 [cs.PL] 25 Jan 2023

摘要。 本文讨论的总体问题是长期存在的程序正确性问题，尤其是描述并行执行进程的程序。我们提出一种新方法，用于证明高层迁移系统规范的并行实现的正确性。该方法所依据的实现语言基于活动（或并发）对象模型。该方法通过程序语义所对应的转移系统与正确性规范所描述的转移系统之间的模拟关系来定义正确性。通过利用本文所考察的特定活动对象模型的全局合流性质，该模拟关系本身抽象掉了并行进程的细粒度交错执行。

作为概念验证，我们将所提出的方法应用于多核内存系统并行模拟器的正确性验证。

1 引言

计算机科学长期面临的一项挑战，是对程序进行形式化规范和验证，尤其是并行程序，例如多线程 Java 程序。这类程序会导致并行执行进程之间出现复杂的细粒度交错执行，并涉及交互机制（加锁）。

粗略而言，我们可以区分基于逻辑的方法和基于语义的方法来确立程序正确性。基于逻辑的方法使用断言来表达行为性质，并生成用于验证这些性质的证明条件；这些条件通常通过交互式定理证明来解决。这些方法适用于无限状态系统，也适用于实践中使用的实际程序（例如，参见 [19]，其中验证了 Java 集合框架中排序程序 TimSort 的一个修正版）。使用逻辑的主要复杂性之一，源于不变性质规范本身的复杂性，以及对定理证明器的交互式使用。另一方面，模型检测基于对程序执行进行自动化状态空间探索。通常，这种方法仅适用于有限状态系统，并且需要采用合适的抽象技术来应对状态空间爆炸问题。

本文的主要贡献是提出了一种新方法，可用迁移系统规范 （简称 TSS）[25,13,26]来描述并行程序整体行为的一种抽象。要验证并行程序是否满足这样的正确性规范，需要在描述并行程序语义的转移系统与 TSS 描述的系统之间建立模拟关系 。

我们的方法支持一种分两步证明并行程序正确性的一般方法：

1. 使用高级形式模型验证全局行为性质。该模型抽象掉目标语言并发模型的复杂性，以便对全局性质进行归纳证明。 2. 通过模拟关系，证明目标语言中的并行实现相对于高级模型是正确的。

迁移系统规范能够以面向语法、组合式的方式，利用描述局部迁移及其组合的推理规则，对系统整体行为进行形式化描述。进程同步可以通过抽象方式表达，例如，将系统状态条件和迁移关系上的可达性条件作为前提，并对并行迁移进行标签同步。这种高层次的抽象大大简化了系统属性的验证。TSS 作为定义语言语义和研究语言元理论的形式化方法广为人知，同时也非常适合描述特定系统，以便研究可达性或状态不变性等问题。

第二步，我们需要一种具有形式语义（例如通过 TSS 形式化定义）的实现语言，以便能够形式化地建立模拟关系。本文选择了活动对象语言 ABS [27,48] （ABS 是抽象行为规范的缩写）。ABS 语言的语义由 TSS [29] 形式化定义，并通过 Erlang、Haskell 和 Java 中的后端[4] 实现，这些后端均支持并行执行。该语言已在多个欧盟项目的背景下开发并应用，例如欧盟第七框架计划项目 HATS[5] （基于形式模型的高适应性可信软件）和 ENVISAGE[6] （虚拟化服务工程）。在这些项目中，ABS 得到了扩展，并成功应用于软件产品族 [17]以及部署在云 [31]上的软件服务的形式化建模与分析。ABS工具套件 [20,6,35,2,32,4,21,34] 还进一步应用于多个案例研究，涉及基于云的框架 [53,40,30,39,3], 、铁路运营 [33] 和计算生物学。[7]

主动对象的并行执行（参见 [18] 关于主动对象语言的综述）是通过异步方法调用，将方法执行与方法调用解耦后直接产生的结果。ABS 语言进一步将主动对象局部状态的严格封装与用于其方法执行协作式调度的显式语言构造相结合。由于 ABS 语言专为描述分布式系统而设计，因此它不考虑方法调用的生成顺序。

在模拟关系的定义中，协作式调度允许主动对象内的方法交错执行，其粒度可与相应 TSS 的转移规则相匹配。此外，ABS 中主动对象的并行执行满足全局合流性质，因此能够以语法制导的方式在局部表达模拟关系的证明条件，而不必考虑方法执行的细粒度交错。

作为概念验证，我们通过将该方法应用于多核内存系统的并行模拟器来进行介绍。这些内存系统通常使用缓存来避免主存数据访问的瓶颈，但缓存会引入数据复制，因此需要协议来确保一致性。尽管数据复制通常对程序员不可见，但程序与这些副本交互的方式会通过移动数据以维持一致性，在很大程度上影响性能。为了开发、测试和优化适用于多核架构的软件，我们需要底层内存系统的正确、可执行模型。文献 [9,10], 描述了多核内存系统的 TSS 及其缓存一致性正确性证明，并给出了基于重写逻辑系统 Maude [15] 的原型实现。然而，这种相当直接的 TSS 实现并不适合模拟大型系统。因此，本文介绍一种基于 ABS 语言主动对象模型的并行实现，并将我们的方法用于证明其正确性。

本文扩展了 [7] ，该文介绍了这一用例的第一个版本。扩展内容包括：将用 TSS 规则注释 ABS 程序这一新颖想法的形式化，以及在这些注释的形式语义（和验证）中运用 ABS 语义的全局合流性质。由于缺少对 ABS 程序与 TSS 之间模拟关系的这种高层规范， [7] 中的 ABS 实现基本上独立于 TSS 开发，这大大增加了正确性证明的难度。相比之下，应用我们的新方法论促使我们对 [7], 中描述的 ABS 实现进行了重大重构，体现了正确性设计开发方法论。

本文结构下一节将介绍 ABS 语言的主要概念，第 3 节将介绍将转移规则用作 ABS 程序注释的方法。第4节将介绍多核 TSS 的运行时语法，第5节将讨论其 ABS 实现。随后，第6节将介绍正确性证明。第 7 节讨论相关工作，第 8 节则总结一些一般性结论并讨论 future 工作。

2 ABS：具有协作式并发的 Actor

ABS 是一种用于设计、验证和执行并发软件的建模语言 [27,48]。核心语言结合了 Java 的语法和面向对象风格，以及演员模型的并发机制 [28], ，由此形成了活动对象，它们通过异步方法调用和协作式调度来解耦通信与同步 [18]。异步方法调用会在被调用的（活动）对象中生成进程（用于执行被调用的方法），且不会在调用者与被调用者之间引入任何同步。相反，不同对象之间的同步通过（隐式）future 实现，调用者和被调用者可以借助它

[4] https://abs-models.org/[5]
https://cordis.europa.eu/project/id/231620[6]
https://cordis.europa.eu/project/id/610582[7]
https://www.compugene.tu-darmstadt.de

指令	含义
新建 C	创建类 C 的实例
分支选择 (e){p_1=> $s_1 \cdots p_n$=> s_n}	模式匹配
等待 b	在布尔条件下挂起
等待 e! m($e_1, \ldots, e_n$)	在异步调用终止时挂起
e!m($e_1, \ldots, e_n$)	非阻塞异步调用
e.m($e_1, \ldots, e_n$)	阻塞式同步调用
m($e_1, \ldots, e_n$)	内联（递归）自身调用

表 1. 本文中使用的 ABS 基本指令。这里，b 是一个布尔表达式，e 和 e_i 表示表达式。

独立地在不同时间进行同步。对象内部不同进程之间的同步则通过协作式调度来描述。进程可以通过显式挂起点让另一个进程获得调度；在挂起点重新调度可能取决于 future 的解析，也可能取决于布尔条件。这种机制使 ABS 能够非常精确地描述不同进程的交错执行。

用于同步和通信的命令式层由函数式层补充，后者用于对对象的内部数据进行计算。函数式层结合了参数化代数数据类型（代数数据类型）和一种带有分情况处理与模式匹配的简单函数式语言。ABS 包含一个库，其中提供预定义类型（整数、布尔值等）和参数化数据类型（列表、集合、映射等）。所有其他类型和函数均由用户定义。

下面将结合一些通用同步模式，解释本文使用的基本 ABS 指令（如图1所示）。

2.1 同步模式

我们将讨论 ABS 中基本锁定机制、原子操作以及用于全局同步的广播机制（使用屏障）的编码。

锁ABS 中异步方法调用和协作式调度的基本机制，可以通过一个简单的 Lock 类代码示例来说明（图1）。它使用布尔值条件上的 **await** 语句来模拟二元信号量，从而确保对共享资源“锁”的互斥访问；该资源被建模为 Lock 类的一个实例（通过执行表达式 **new**Lock 动态创建）。具体而言，take_lock 方法的执行会被**await**未锁定语句挂起。此语句释放控制权，允许在 Lock 对象中调度其他（已启用的）进程。当 Lock 对象内部的局部条件未锁定变为真时，生成的 Lock 对象内的 take_lock 进程将竞争执行。随后，被调度的进程会将未锁定设为假，然后终止并返回。

```
类锁 {
  布尔值 未锁定 =真;

  单元 获取_锁{
    等待未锁定;
    未锁定 =假;
  }

  单元 释放_锁{
    未锁定 =真;
  }
}
```

图1。在 ABS 中使用布尔值上的 await 实现锁。

一般而言，由挂起点 定义的**await**语句决定了对象的进程交错执行的粒度。该语句
await lock!take_lock() 只会使发起调用的进程挂起（并释放调用者对象中的控制权），直至 take_lock 返回。相比之下，在 ABS 中，同步 调用 lock.take_lock() 会为锁对象执行 take_lock() 方法创建一个进程，并阻塞调用者对象的所有进程，直至该方法返回。

原子操作 ABS 的并发交错执行模型允许以简单且高层次的方式实现原子操作。例如，图 2 展示了测试并设置指令的一种通用 ABS 实现 [5]，，其中并发模型保证，假设本地*/*test(input)*/* 和 */*set*/* 指令不涉及挂起点，它们就不会交错执行，因此可以视为在单个原子操作中执行。在 ABS 中，测试指令可以

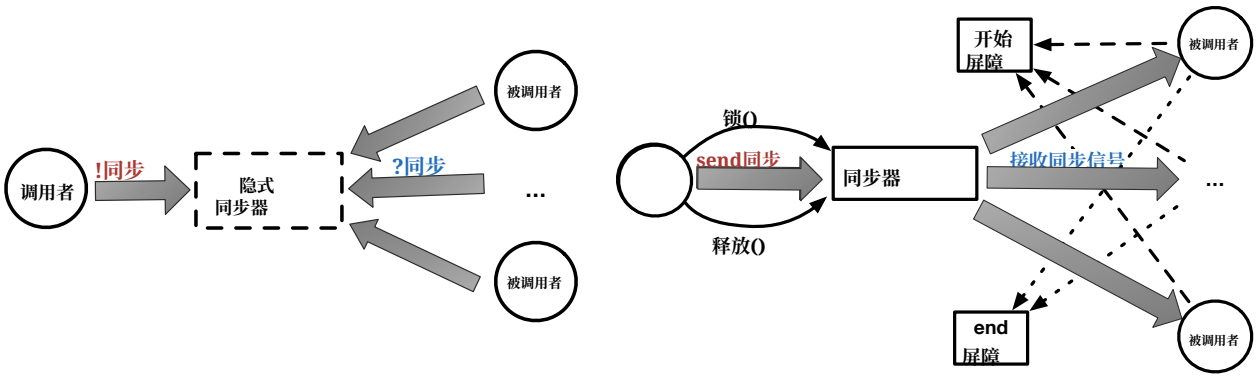
```
布尔值 TestandSet (/*输入*/)
```

图 2. ABS 中的测试并设置模式。

得到实现 使用**分支选择**指令，对一个表达式求值，并将结果值 ue

与不同分支中的模式 p 进行匹配。该指令主要用于对本文讨论的 ABS程序中使用的代数数据类型进行模式匹配。在最简单的情况下，此模式可以替换为**if-else**指令。这种原子模式的实例可见于图11和图14中的方法 re move_inv和 swap。

广播同步 图3a展示了如何在带标签的TSS中仅通过匹配标签来实现广播同步（第4节详细介绍了一个示例），从而抽象掉隐式多方同步器的实现细节。另一方面，在ABS等编程语言中，多方标签同步需要显式编程；图3b展示了图4中ABS 实现的架构。



a）带标签的TSS中的广播同步。

b）使用显式同步器的广播同步以及ABS 模型中的屏障。

图3。 TSS和ABS中的广播同步模式。

classBroadcast 类充当模板（或设计模式），用于实现由 IBroadcast 接口指定的、其各个实例之间的广播机制。broadcastSync 方法封装了 Broadcast 实例之间的一种同步协议，该协议使用额外的 Synchroniser 类和 Barrier 类。此协议包含对 Synchronise 类某个实例（记为 sync）的 sendSync 方法进行同步调用；该实例随后会异步调用 Broadcast 实例的 network 集合中所存对象的 receiveSync 方法，但不包括正在执行 broadcastSync 方法的调用者对象。我们不限定 sync 对象是作为参数传递给 broadcastSync 方法，还是属于任一 Broadcast 实例的局部状态。receivers 中的对象通过调用新建的 Barrier 类实例 start 和 end 的 synchronise 方法，对 receiveSync 方法指定的局部计算进行同步。也就是说，start 和 end 屏障执行该方法，以同步 receivers 中各对象开始和终止执行 receiveSync 方法的时刻，并同步 sendSync 方法本身的终止。这通过对 receivers 中已调用 synchronise 方法的对象数量进行倒计数来实现；对于结束屏障，计数还要加一。开始屏障的 synchronise 方法以异步方式调用（第 15 行），并引入一个释放点，以避免可能出现的死锁：某个尚未调用开始屏障 synchronize 方法的对象，可能会阻塞在对某个已（同步）调用开始屏障 synchronize 方法的对象所进行的同步方法调用上。另一方面，对结束屏障的相应调用是同步的，以确保 receivers 中的所有对象都已完成各自的局部计算。在结束屏障上对同步器对象进行额外同步，可确保 sendSync 方法的调用者也会一直阻塞，直到 receiveSync 方法指定的所有局部计算均已完成。

ABS 中的对象是输入启用的，因此始终可以调用对象的方法。在我们的实现中，如果多个对象并行启动该协议，这种机制可能导致不一致状态。为确保协议开始时对同步器的互斥访问，我们在同步器协议中加入一把锁，调用者必须先获取该锁，才能调用 sendSync，并在调用完成后释放该锁。由此实现的对同步器的互斥访问，可确保其消息池中至多包含一个对 sendSync 方法的调用。

```
IBroadcast 接口 {
布尔值 同步广播(...);
单元 同步接收 (IBarrier 开始, IBarrier end, ...)}

类 广播 实现 IBroadcast,...{
布尔值 同步广播(...){
    布尔值 signal=假;
    等待 同步!lock();
    如果 /*测试*/ {sync.sendSync(this,...); /*设置*/;signal=真; }
    sync.释放();
    返回 signal
}

单元 receiveSync(IBarrier 开始, IBarrier end, ...) {
     等待 开始!同步();
   /*一些本地计算*/;
    结束.同步(); }
...
}

类 同步器 (集合<IBroadcast>network) 实现 ISynchroniser {
布尔值 未锁定 = 真;
单元 锁(){ 等待 未锁定; 未锁定 = 假; }
单元 释放(){未锁定 = 真; }
单元 sendSync(IBroadcast 调用者,...) {
      集合<IBroadcast>接收方 =移除(network,调用者);
      整数 nrrecs=大小(接收方);
      IBarrier 开始 =新建 屏障(nrrecs);
      IBarrier 结束 = 新建 屏障(nrrecs+1);
      对每个 （接收方 无效状态 接收方）{ 接收方!同步接收(开始,结束,...); }
      结束.同步();}
 }


类屏障(整数 参与者) 实现 IBarrier {
  单元 同步() { 参与者 = 参与者 − 1; 等待 （参与者 == 0）； }
}
```

图4。 ABS 中的全局同步模式。

2.2 语义

ABS 是一种形式化定义的语言 [29]；事实上，其（操作）语义由一个 TSS 定义，使我们能够对 ABS 程序的执行进行形式化推理。ABS 模型的语义可以用全局配置之间的转移关系来描述。全局配置是对象配置的一个（有限）集合。对象配置是如下形式的元组：$\langle oid\ \sigma, p, Q\rangle$，其中 oid 表示对象的唯一标识，σ 为对象的实例变量（字段）赋值，p 表示当前正在执行的进程，Q 表示一组（挂起的）进程（即对象的“队列”）。进程是一个闭包（τ, S），由语句 S 的局部变量赋值 τ 构成。有关在 ABS 中推导全局配置之间转换 $G \to G'$ 的 TSS 详情，请参阅 [29] 。

尽管任一时刻活动对象中只能有一个控制线程执行，但协作式调度允许不同线程在代码中明确声明的点交错执行，即 **await**语句。当当前正在执行的进程被 **await** 语句挂起时，系统会调度另一个（已启用的）进程。对象的字段受到保护：对字段的任何非局部（对象外部）读取或写入，都会通过方法调用显式进行，以缓解竞态条件或避免大量使用显式互斥机制（锁）。

由于活动对象仅通过方法调用进行交互，且进程以非确定性方式调度，这就抽象了由方法调用生成进程的顺序，因此 ABS 语义

满足以下 全局合流性质（另见 [7,54]），该性质允许交换属于不同对象的进程的连续局部计算步骤 骤，这些进程属于不同的对象。

定理 1（全局合流）。对于描述不同对象的进程执行步骤的任意两个转换 $G_1 \to G_2$ 和 $G_1 \to G_3$，存在一个全局配置 G_4，使得 $G_2 \to G_4$ 和 $G_3 \to G_4$。

上述全局合流性质的一个重要推论是，我们可以通过重新排列 ABS 计算中的执行步骤，限制进程之间的全局交错执行；本文的主要结果正是建立在这一点之上。具体而言，考虑同步通信的一般语义性质，以及 ABS 中锁和广播同步的实现后，我们可以限制 ABS 模型的交错语义，具体如下所述。

由于在 ABS 中，对另一个 对象的方法进行同步调用会阻塞调用者（对象）的所有进程，因此全局合流性质允许进一步限制 ABS 进程的交错执行，使调用者进程在同步方法调用终止后立即 恢复执行。

值得注意的是，一般来说，我们不能 假设在 ABS 中被同步调用的方法也会立即得到调度并开始执行，因为这样会丢弃被调用者的其他进程的执行。

全局合流性质还允许我们抽象掉上述 ABS 实现全局（广播）同步模式时的内部计算步骤，因为它允许对由广播 方法生成的进程进行调度，使该方法的执行不与任何其他进程交错。

我们可以用以下稳定 配置的概念形式化上述内容。

定义 1（稳定配置）。如果对象配置中待执行的语句表示一个异步 调用的方法已终止（我们假设有一种专门表示此类终止的运行时语法），或者该语句以对另一个对象 的同步调用或**await** 语句开头，则该对象配置是稳定的。如果全局 *ABS* 配置中的所有对象配置都是稳定的，则该全局 *ABS* 配置是稳定的。

请注意，同步自调用通过内联执行，因此不构成交错点。

下文中，$G \Rightarrow G'$ 表示转移关系，它描述从全局稳定配置 G 开始执行并到达下一个全局稳定配置 G' 的过程（其间不经过其他全局稳定配置）。我们区分以下三种情况：

1. 转换 $G \Rightarrow G'$ 描述单个对象执行方法的局部 过程。 2. 转换 $G \Rightarrow G'$ 描述同步方法调用中调用者与被调用者之间的会合，具体而言，是被调用方法的终止执行，随后 挂起的调用恢复执行。 3. 转换 $G \Rightarrow G'$ 描述执行广播方法所产生的效果，因此描述了不同对象之间的全局 同步。

这种 ABS 的粗粒度交错语义构成了证明 TSS 规范的 ABS 实现正确性的一般方法的基础，下一节将对此进行介绍。

3 一般方法论

3.1 使用 TSS 规则标注 ABS

关于迁移系统规范的概述，请参见 [25]。开发抽象 TSS 的 ABS 实现的一般方法论基于第 2 节中所述的粗粒度交错执行（由转移关系 $\Rightarrow$ 表示）：它使我们能够专注于设计实现各个转换规则的局部顺序代码。这体现在以下将转换规则用作 ABS 代码的规范形式。一个条件转移规则 b: R 由 ABS 中的局部布尔条件b和一个转换规则的名称 R组成。我们使用稳定点的条件转移规则序列 b_1： R_1; …; b_n： R_n 进行标注。方法定义的稳定点表示其方法体，或方法体中以外部同步调用或**await** 语句开头的子语句。其思路是，将每个 b_i 求值为一个条件，用于识别一条路径，该路径从带注释的稳定点通向下一个稳定点或终止。此路径的执行应与应用相应转换规则R_i相对应。这种对应关系涉及一种模拟关系，将在下文描述。

条件转移规则序列 b_1： R_1; …; b_n： R_n 按从左到右的顺序进行求值，也就是说，从左侧开始，第一个求值结果为真的布尔条件所对应的转移规则适用。当所有

布尔条件均为假时，意味着不存在从带注释的稳定点到下一个稳定点或终止的任何路径对应的转移规则（在模拟关系中，所有这些路径都对应于一次“静默”转换）。作为特殊情况，我们规定，对于从任何稳定点（该稳定点没有关联的注释）通往下一个稳定点（或终止）的路径，不存在相应的转移规则。多核内存系统的ABS代码中注释的使用见第5.2节。

3.2 实现的正确性

相对于给定 TSS，ABS 实现的正确性可以通过描述 ABS 实现语义的转移系统与描述 TSS 的转移系统之间的模拟关系来确立。用（条件）TSS 规则对 ABS代码进行注释，提供了模拟关系的高层描述，说明哪些规则与 ABS 代码从一个稳定点执行到下一个稳定点（或终止）相对应。在这一高层描述的基础上，我们定义 ABS 配置与 TSS 运行时状态之间的模拟关系。该模拟关系由抽象函数 α 定义，它将每个稳定的全局 ABS 配置 G 映射为一个行为等价的 TSS 配置 $\alpha(G)$（见第6节）。

我们将模拟关系限制在可达的 ABS 配置上。如果 $G_0 \Rightarrow^* G$，则 ABS程序的配置 G 是可达的，其中 $G_0 \Rightarrow^* G$表示某个初始 配置 G_0。在 ABS 多核程序的初始配置中，所有进程队列均为空，唯一活动的进程是即将执行各核心 run 方法的进程。这一限制使我们能够使用 ABS 语义的一些一般性质；例如，同步调用返回时，调用对象的局部状态并未改变。

现在，只需证明以下定理成立，即可表明一个 ABS程序是某个 TSS 规范的正确实现，其中给定一个抽象函数 α:

定义 2（正确性）。给定一个 *ABS*程序和一个 *TSS*，设 α 为从 *ABS*程序的配置到 *TSS* 配置的抽象函数。如果对于 *ABS*程序的任意可达配置 G 及转换 $G \Rightarrow G'$，都有 $\alpha(G) = \alpha(G')$ 或$\alpha(G) \to \alpha(G')$，则该 *ABS*程序是 *TSS* 的正确实现。

由于ABS语义具有一般的合流性质，要证明 α 是一种模拟关系，因此只需根据抽象函数 α 验证方法的注释即可。基本思想是，对于每个转换 $G \Rightarrow G'$（该转换由从一个稳定点执行到下一个稳定点或终止所产生），我们必须证明，通过应用与初始稳定点关联的已启用 TSS规则，$\alpha(G')$ 可由 $\alpha(G)$ 得到。如果没有启用任何 TSS规则，则执行一个“静默”步骤，即 $\alpha(G) = \alpha(G')$。

4 用于多核内存系统的 TSS

可利用模拟器（例如，[11,14,41,44]）探索在多核内存系统之上运行的程序的设计决策。Bijo 等人开发了一个用于多核内存系统的 TSS [9,10]。我们将以该 TSS 为起点，研究如何开发一个实现该 TSS 的并行模拟器，并利用这一开发过程讨论我们的方法论细节。我们首先介绍多核内存系统的主要概念，然后考察如何以 TSS 的形式将其形式化。

多核内存系统由包含待执行任务的核心、主存中的数据布局（指示数据分配位置），以及由具有私有多级缓存和共享内存的核心组成的系统架构构成（见图5）。此类系统的参数包括核心数量、缓存的数量和大小、相联度以及替换策略。数据被组织为在缓存与主存之间移动的块。

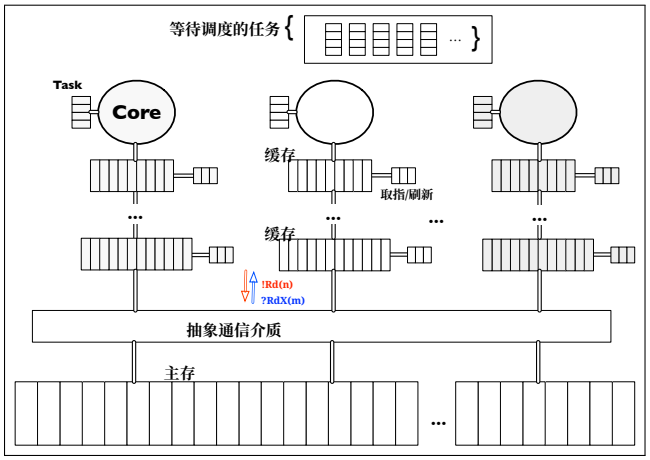


图5。 多核内存系统的抽象模型。

为简单起见，w 抽象掉内存块中的数据内容，假设缓存行的大小与 d

语法
类别。 定义。

$$
\begin{array}{lll}
cid \in CoreId & cf \in Config & ::= \langle \underline{CR}, \underline{Ca}, M \rangle \\
caid \in CacheId & CR \in Core & ::= cid \bullet rst \\
n \in Address & Ca \in Cache & ::= caid \bullet M \bullet dst \\
 & st \in Status & ::= \{mo, sh, inv\} \\
 & dap \in AccessPtns & ::= \varepsilon \mid dap; dap \mid \textbf{读取}(n) \mid \textbf{写入}(n) \\
 & rst \in RunLang & ::= dap \mid rst; rst \mid \textbf{ReadBl}(n) \mid \textbf{WriteBl}(n) \\
 & dst \in DataLang & ::= \varepsilon \mid dst + dst \mid \textbf{获取}(n) \mid \textbf{fetchBl}(n) \\
 & & \quad \mid \textbf{fetchW}(n, n') \mid \textbf{刷新}(n)
\end{array}
$$

图 6. 运行时配置的语法，其中上划线表示集合（例如，CR）。

主存中的内存块大小相同，并且对内存块的本地引用直接由相应的内存地址表示；通过主存将内存块从一个核心的缓存传输到另一个核心的缓存。因此，核心中执行的任务被表示为数据访问模式，从而抽象掉其计算内容。

任务在核心上执行时，需要将内存块从主存传输到最近的缓存。每个缓存都有一组指令，用于在缓存之间以及缓存与主存之间移动内存块。内存系统中可能存在同一内存块的多个副本。使用标准缓存一致性协议 MSI（例如，[51]）确保内存块不同副本之间的一致性；根据该协议，缓存行可以处于已修改、共享或无效状态。已修改 的缓存行包含该内存块的最新值，因此所有其他副本（包括主存中的副本）均为无效 状态。共享 缓存行表示该块的所有副本保持一致。协议消息会广播给各个核心。广播的具体实现细节（例如通过网格或环形网络）可以抽象为抽象通信介质。按照标准术语，Rd 消息请求对内存块进行读取 访问，而RdX 消息请求对内存块进行独占读取 访问。后者会使其他缓存中同一块的副本失效，从而提供写访问。

我们总结了 MSI 协议下缓存一致性的操作特性。要访问内存块 n 中的数据，核心会在本地缓存中查找 n。如果未找到处于共享状态或修改状态的 n，就会向其他核心和主存广播读请求$!Rd(n)$。当该块在主存中可用时，缓存即可获取 该块。如果缓存已满，就需要驱逐一个块，以腾出可用空间。要写入块 n，本地缓存中的 n 必须处于共享状态或修改状态；如果它处于共享状态，则会广播失效请求 $!RdX(n)$，以获得互斥访问。如果某个缓存中处于修改状态的块 n 收到读请求 $?Rd(n)$，它会将该块刷新 到主存；如果某个缓存中处于共享状态的块 n 收到失效请求 $?RdX(n)$，则该缓存行将被置为无效；否则，请求将被丢弃。在抽象模型中，读取请求和失效请求会被即时广播，这反映出通信介质上的信号传输速度比从主存读取或向主存写入数据快几个数量级。

4.1 多核内存系统的 TSS

多核 TSS 描述核心、缓存和主存之间的交互。它还包含用于建模瞬时广播的标记迁移。一般而言，多核 TSS 的模型是一个转移系统。我们也将多核 TSS 的模型称为多核内存系统（简称 MMS）；该模型的参数为核心和缓存的数量。多核 TSS [9,10] 被证明能够保证数据一致性和缓存一致性的正确性属性（例如，参见 [16,52]），包括保持每个核心的程序顺序、避免数据竞争，以及永不访问过时数据。

我们概述多核 TSS 简化版本的主要方面，以便聚焦于正确分布式实现所面临的主要挑战。运行时语法见图 6。配置 cf 是一个由主存 M、核心 CR、缓存 $\underline{Ca}$ 组成的元组（我们不考虑待调度的任务）。标识符为 cid 的核心（$cid \bullet rst$）执行运行时语句 rst。标识符为 $caid$ 的缓存（$caid \bullet M \bullet dst$）具有本地缓存存储器 M 和数据指令dst。我们假设缓存标识符 $caid$ 编码了该缓存所属核心的 cid 及其在缓存层次结构中的级别。我们使用状态$_\perp$ 表示状态标签集合 $\{mo, sh, inv\}$ 扩展后包含未定义值 $\perp$ 的集合。因此，内存 M: 地址 $\to$ 状态$_\perp$将地址 n 映射为状态标签st 或 $\perp$；当地址为 n 的内存块在 M 中找不到时，即为 $\perp$。

数据访问模式dap 将任务建模为由有限序列组成的**读取**(n) 和**写入**(n) 操作，以访问 n（即，我们对顺序组合、非确定性选择、重复和任务创建中的控制流操作进行抽象）。空访问模式表示为 ε。核心执行运行时语句rst,

这些语句通过 readBl(n) 和 writeBl(n) 扩展 dap，以便在等待数据时阻塞执行。缓存从多重集 dst 中执行数据指令，以获取或刷新地址为 n 的内存块；其中，fetch(n) 获取地址为 n 的内存块，fetchBl(n) 在等待数据时阻塞执行，fetchW(n, n') 等待内存块 n' 被刷新后再获取 n（当缓存已满时这是必需的），而 flush(n) 刷新一个内存块。

不同核心的主存与缓存之间的连接由一种抽象通信介质建模，该介质允许将来自一个缓存的消息通过并行即时广播传输到其他缓存和主存。TSS 中通过转换上的标签匹配来描述抽象通信介质中的通信。对于任意地址 n，形式为 $!Rd(n)$ 或 $!RdX(n)$ 的输出会被广播，并与其对偶形式 $?Rd(n)$ 或 $?RdX(n)$ 匹配。模型的语法将在 [9,10] 中进一步详细说明。有关转换规则的完整概述，请参阅 A。下一节将介绍这些规则，并在讨论其 ABS 实现时逐步展开。

T转换规则中使用了以下辅助函数，给定一个缓存标识符 cai d:

– $cid(caid)$ 返回该缓存所属核心的标识符；– $lid(caid)$ 给出该缓存在内存层次结构中的层级；– $first(caid)$ 在 $lid(caid) = 1$ 时为真，否则为假；– $last(caid)$ 在 $lid(caid) = l$ where l 时为真，其中 l 是层次结构中的缓存数量，否则为假；– $status(M, n)$ 返回内存 M 中块 n 的状态；如果在 M 中找不到该块，则返回 $\perp$；以及– $select(M, n)$ 根据缓存相联度（例如随机、组相联或直接映射）和替换策略（例如随机或最近最少使用），确定块 n 应放入缓存的哪个地址。

5 多核内存系统的 ABS 模型

本节介绍如何将多核 TSS 转换为 ABS 模型 [8]。我们阐述这两个模型之间的结构对应关系和行为对应关系。

5.1 结构对应关系

多核 TSS 的运行时语法通过 ABS 中的类、用户定义数据类型和类型同义词来表示，如图 7–9 所示。ABS 配置由类实例构成，以反映各核心及其对应的缓存层级和主存。对象标识符确保名称唯一，而对象引用则用于描述核心与缓存之间的关联。这些引用与多核 TSS 的命名方案一一对应。

多核 TSS 中的核心 $cid \bullet rst$ 对应于 ABS 中类 Core 的一个实例，其中类型为 RstList（定义见图 9）的字段 currentTask 表示当前的运行时语句列表。类 Core 的每个实例还持有对一级缓存的引用。我们做出的一个重要设计决策是，将（多核 TSS 中核心的）运行时语句 rst 表示为代数数据类型（见图 9）。随后，ABS 中的核心通过处理这些运行时语句来驱动仿真，而处理过程通常需要一级缓存的信息。另一种做法是，ABS 中的核心可以调用一级缓存的相应方法，将运行时语句的处理委托给它。然而，后一种做法会使所需的回调变得复杂。

多核 TSS 中的一个缓存 $caid \bullet M \bullet dst$ 对应于类 Cache 的一个实例，该类具有一个类参数 nextLevel，用于保存对下一级缓存的引用，以及一个字段 cacheMemory，用于建模该缓存的内存 M（类型为 MemMap，见图 9）。缓存的数据指令多重集 dst（见图 6）在 ABS 中由缓存对象的消息池中相应的进程 表示。如果 nextLevel 的值为 Nothing，则该对象表示末级缓存（在多核 TSS 中，使用谓词 $last$ 来标识末级缓存）。

此外，多核 TSS 中使用的、带有标签 $!Rd(n)$ 和 $!RdX(n)$ 的全局同步 ABS 实现，基于图4所述的全局同步模式。不过，我们没有通过额外参数来区分这两个标签，而是引入了两个相应的广播接口： ¨

[8] ABS 模 型的多核内存系统可在 https://abs-models.org/documentatio 中找到 n/ 示例/多 核_内存/

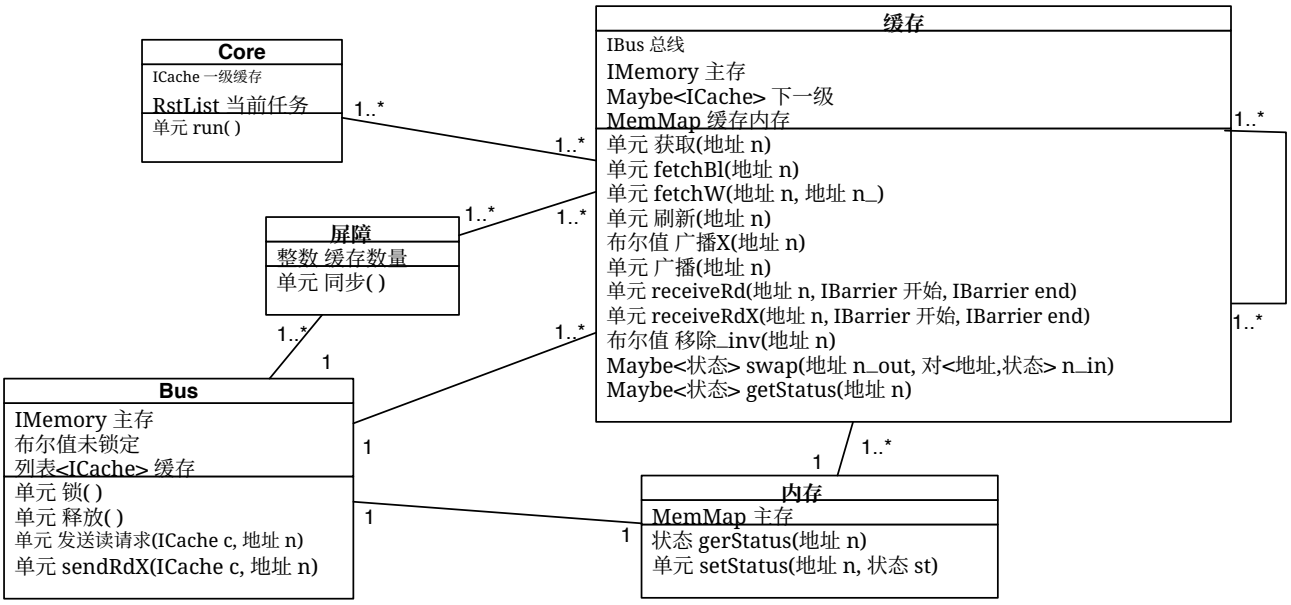


图 7. ABS 模型的类图。

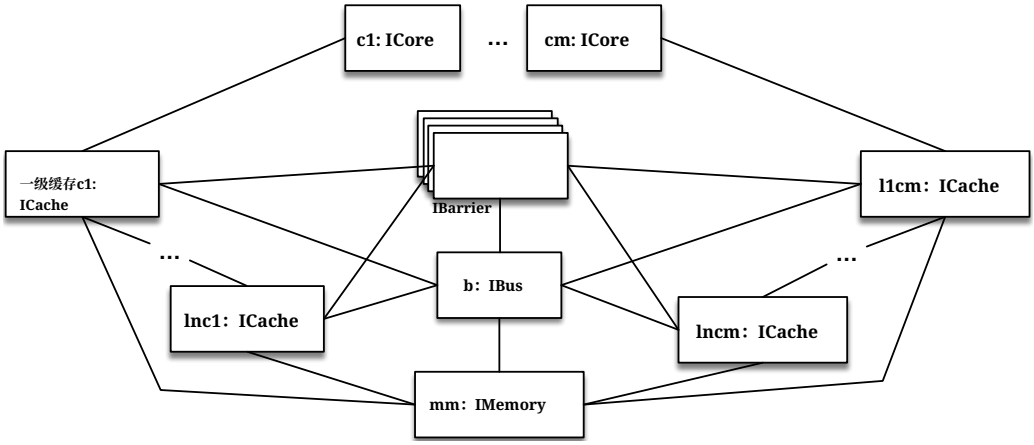


图 8. 初始配置的对象图。

```
数据 Rst = 读取(地址) | ReadBl(地址) | 写入(地址) | WriteBl(地址);数据 状态
```

图 9. 多核内存系统模型的抽象数据类型。

```
IBroadcast 接口 { 布尔值 broadcast(...); 单元
```

```
单元 运行 {
  如果当前任务！=空值 {
  分支选择(当前任务) {
    Cons(rst, 剩余部分) =>
    分支选择 (rst){
      读取(n) =>{
          已移除 = 一级缓存.移除 _inv(n); // 已移除==真： PrRd2; 已移除==假： PrRd1
          如果 (已移除){
            一级缓存!获取(n);
            当前任务 = Cons(读取块(n),剩余部分); }
          否则 {当前任务 = 剩余部分; } }
       ReadBl(n) =>{
         状态 = 一级缓存.getStatus(n); // 状态！=无： PrRd3
        如果 (状态 != 无效状态) 当前任务 = Cons(读取(n),剩余部分); }
      写入(n) =>{
        状态 = 一级缓存.getStatus(n); // 状态==Just(修改状态): PrWr1
        分支选择 (状态) {
          Just(修改状态) => {当前任务 = 剩余部分; }
          Just(共享状态) =>{
            布尔值 res = 一级缓存.广播X(n); //res==真: PrWr2/SynchX
            如果 (res) {当前任务 = 剩余部分; } }
          _ => { 布尔值 已移除 =一级缓存.移除 _无效状态(n); // 已移除==真: PrWr3
                如果 (已移除){一级缓存!获取(n); 当前任务 = Cons(WriteBl(n),剩余部分); } } } }
      WriteBl(n) =>{
        Maybe<状态> 状态 =一级缓存.getStatus(n); // 状态！=无: PrWr4
        如果 (状态 != 无)
          当前任务 = Cons(Write(n),剩余部分); }
    }
  此！run();}
}
```

图10. 带注释的 run 方法。

随后，类 Cache 按照图4中类 Broadcast 的模板，提供了这两个接口的实现。另一方面，ABS 类 Bus 遵循同步器类的模板，其中 sendSync 方法有 sendRd 和 sendRdX 两个版本。

图8中的对象图展示了一个初始配置，与图5所示的配置相对应。

5.2 行为对应

接下来，我们讨论多核 TSS 转移规则的 ABS 实现，以及第 2 节中描述的 ABS 同步模式。我们注意到，ABS 中异步方法调用 与协作式调度 的结合至关重要，因为多核 TSS 固有的交错执行 要求对象能够以受控方式在执行方法的同时处理其他请求；例如，缓存需要在等待获取成功时刷新内存块。

带注释的 ABS 多核实现 Core 和 Cache 类带来了主要的实现挑战。这里我们非正式地说明 Core 类的 run 方法（图10）的实现（这是该类唯一的方法），并从其注释的角度进行解释（如第 3.1 节所介绍）。第 6 节将介绍这些注释的形式语义，将其作为模拟关系的高层描述，并证明 Cache 类的正确性。

run 方法可能会对图 11 中的辅助方法进行同步调用。方法 remove_inv 实例化了图 2 中的测试并设置模式。方法 broadcastX 是第 2 节图4所述全局同步模式的一个实例。全局同步总线的 sendRdX 方法会使用第 2 节所述的屏障同步，异步调用所有缓存（调用该方法的缓存除外）的 receiveRdX 方法，参见图 12。

由于稳 定点在 run 方法开头，且没有关联的注释，根据定义
（参见第 3.1 节）对于从起始处到下一个稳定点（或到终止）的任意路径，都存在 *not*

```
Maybe<状态> getStatus(地址 n) { 返回 查找(缓存内存,n); }

布尔值 移除_无效状态(地址 n){
  布尔值 answer =假;
  分支选择 (查找(缓存内存,n)) {
    无 => { 回答 = 真; }
    Just(无效状态) =>{ 缓存内存 = 移除键(缓存内存,n); 回答 = 真; }
    _ => 跳过; }
  返回 回答; }

布尔值 广播X(地址 n) {
  布尔值 res = 假;
  等待 总线!锁(); //(查找(缓存内存,n) ==Just(共享状态)): PrWr2/同步
  如果 (查找(缓存内存,n) ==Just(共享状态)){
    总线.发送读请求X(此, n);
    缓存内存 =放入(缓存内存,n,修改状态);
    结果 = 真; }
    总线.释放();
  返回结果; }
```

图11。 类 C 缓存的 getStatus、移除_inv 和 broadcastX 方法。

```
单元 receiveRdX(地址 n,IBarrier start,IBarrier end) {
  // 查找(缓存内存,n))==Just(共享状态): Invalidate-One-Line;
  // 查找(缓存内存,n))!=Just(共享状态): 忽略单行失效
  等待start!同步();
  分支选择 (查找(缓存内存,n)) {
    Just(共享状态) => {缓存内存 = 放入(缓存内存,n,无效状态); }
    _ => 跳过;
  }
  end.同步(); }
```

图 12. 带注释的 receiveRdX 方法。

一个对应的转移规则（属于多核 TSS）。例如，当 curentTask==Nil 时 run 方法终止，这种情况没有对应的转移规则（注意，由于存在结构对应关系，相应的核心也没有运行时语句 rst 可执行）。类似地，从方法起始处到对一级缓存的辅助方法 remove_inv（第 7 行）和 getStatus（第 13、16、25 行）的同步调用为止，这段代码的执行也没有对应的转移规则；除了模式匹配之外，这段代码仅包含调用本身。

注释条件 已移除==真： PrRd_2 (第 7 行) 与同步调用 移除_inv 方法相关，描述了从该方法执行开始，并通过随后 **then**-分支返回的路径，该分支属于后续的 **if**-语句；此路径会导致 run 方法终止（在异步地再次调用自身之后）。根据该注释，此路径的执行对应于 PrRd_2 转移规则：

$$\frac{(\text{PrRd}_2)}{first(caid) = true \quad cid(caid) = c \quad status(M,n) \in \{inv, \bot\}}$$
$$(c \bullet \mathbf{read}(n); rst),\ (caid \bullet M \bullet dst) \rightarrow$$
$$(c \bullet \mathbf{readBl}(n); rst),\ (caid \bullet M[n \mapsto \bot] \bullet dst + \mathbf{fetch}(n))$$

此规则处理核心要读取地址为 n 的内存块，但该内存块未在一级缓存中找到的情况。随后，核心将被阻塞，等待从低级缓存或主存获取该内存块。请注意，remove_inv 方法返回的条件表示一级缓存中该地址的状态未定义或无效。

另一方面，条件 removed==False 描述了这样一条路径：执行后通过**else**分支（第 11 行）返回，并由此终止此次 run 方法调用。

根据注释，这条路径的执行对应于 PRRD_1 转移规则：

$$\frac{(\text{PRRD}_1)}{first(caid) = true \quad cid(caid) = c \quad status(M,n) \in \{sh, mo\}}$$
$$(c \bullet \textbf{read}(n); rst),\ (caid \bullet M \bullet dst) \rightarrow (c \bullet rst),\ (caid \bullet M \bullet dst)$$

此规则涵盖了核心要读取的内存块位于其一级缓存中的情况。注意，remove_inv 方法返回的条件意味着一级缓存中该地址的状态要么是共享状态，要么是修改状态。

接下来，我们考虑同步调用 getStatus 方法的注释= 状态！ 无$_3$: getStatus 方法（第 13 行）。其条件描述了这样一条执行路径：从被调用的getStatus 方法执行并返回，到通过后续run 方法的**then**分支**if**语句（第 14 行）而终止。根据该注释，这条路径的执行对应于PRRD_3 转移规则：

$$\frac{(\text{PRRD}_3)}{first(caid) = true \quad cid(caid) = c \quad n \in dom(M)}$$
$$(c \bullet \textbf{readBl}(n); rst),\ (caid \bullet M \bullet dst) \rightarrow (c \bullet \textbf{read}(n); rst),\ (caid \bullet M \bullet dst)$$

当 n （即要读取的块）位于一级缓存中时，此规则会解除核心的等待状态。另一方面，不存在与条件 status ==Nothing 所描述的执行路径相对应的转移规则。这条路径从被调用的 getStatus 方法执行后直接到达 run 方法的终止，期间局部状态没有更新，例如，currentTask 未更新。换言之，在 ABS 中，readBl(n) 指令的求值会涉及忙等待，直至一级缓存返回的状态有定义。另一种实现方式是同步调用一级缓存的方法，该方法只执行语句**await**lookup(cacheMemory,n)!=Nothing。

对 getStatus 方法的同步调用（第 25 行）的注释涉及该转移规 e

$$\frac{(\text{PRWR}_4)}{first(caid) = true \quad cid(caid) = c \quad n \in dom(M)}$$
$$(c \bullet \textbf{writeBl}(n); rst),\ (caid \bullet M \bullet dst) \rightarrow (c \bullet \textbf{write}(n); rst),\ (caid \bullet M \bullet dst)$$

此注释与第 13 行 getStatus 方法同步调用的注释解释方式类似。当在一级缓存中找到 n （即要写入的块）时，此规则会解除核心的等待状态。

接下来，我们考虑状态==Just(修改状态)： PRWR_1 对getStatus方法的同步调用（第 16 行）的注释。其条件描述了从被调用的getStatus方法的执行，到随后执行**分支选择**语句，再到run 方法终止的执行路径。根据该注释，此路径的执行对应于PRWR_1 转移规则：

$$\frac{(\text{PRWR}_1)}{first(caid) = true \quad cid(caid) = c \quad status(M,n) = mo}$$
$$(c \bullet \textbf{write}(n); rst),\ (caid \bullet M \bullet dst) \rightarrow (c \bullet rst),\ (caid \bullet M \bullet dst)$$

如果在一级缓存中发现内存块 n 处于已修改状态，此规则允许核心向该内存块写入。另一方面，如果条件不成立，根据注释，从被调用的 getStatus 方法执行到下一个稳定点（即对 broadcastX 和 remove inv 方法的同步调用，分别位于第 20 行和第 22 行）的执行路径并不对应任何转换规则。

同步调用res==真： $\text{PRWR}_2/\text{SYNCHX}$ 一级缓存中 广播X方法（第 20 行）的注释条件描述了这样一条路径：从执行 广播X方法开始，接着执行后续的 if 语句，直到 run 方法终止（更新当前任务 并再次异步调用 run 方法之后）。根据该注释，这条路径对应于全局同步规则

$$\frac{(\text{SYNCHX})}{CR \notin \overline{CR_1} \quad CR, \overline{Ca} \xrightarrow{!RdX(n)} CR', \overline{Ca'}}$$
$$\langle \overline{CR_1} \cup \{CR\},\ \overline{Ca},\ M\rangle \rightarrow \langle \overline{CR_1} \cup \{CR'\},\ \overline{Ca'},\ M[n \mapsto inv]\rangle$$

其中，第二个前提是通过连续应用以下规则生成的

$$\frac{Ca_1 \notin \overline{Ca} \quad CR, \overline{Ca} \xrightarrow{!RdX(n)} CR', \overline{Ca'} \quad Ca_1 \xrightarrow{?RdX(n)} Ca_2}{CR, \overline{Ca} \cup \{Ca_1\} \xrightarrow{!RdX(n)} CR', \overline{Ca'} \cup \{Ca_2\}} \text{(SYNCH-DISTX)}$$

后者规则本身由以下规则触发

$$\frac{first(caid) = true \quad cid(caid) = c \quad status(M, n) = sh}{(c \bullet \mathbf{write}(n); rst), (caid \bullet M \bullet dst) \xrightarrow{!RdX(n)} (c \bullet rst), (caid \bullet M[n \mapsto mo] \bullet dst)} \text{(PRWR}_2\text{)}$$

$$\frac{status(M, n) = sh}{caid \bullet M \bullet dst \xrightarrow{?RdX(n)} caid \bullet M[n \mapsto inv] \bullet dst} \text{(INVALIDATE-ONE-LINE)}$$

and

$$\frac{status(M, n) \in \{inv, \bot\}}{caid \bullet M \bullet dst \xrightarrow{?RdX(n)} caid \bullet M \bullet dst} \text{(IGNORE-INVALIDATE-ONE-LINE)}$$

这些规则共同刻画了多核内存系统中用于失效的广播机制。规则 $\mathbf{PrWr}_2$ 对应于核心向内存块 n 写入的情形，该内存块在其一级缓存中被标记为共享状态，因此需要向所有其他缓存广播一条失效消息 $!RdX(n)$。这通过触发全局同步规则 SynchX 和 Synch-DistX 来实现。前者标识广播失效消息的核心 CR，后者则将消息 $?RdX(n)$ 递归传播到其他缓存。根据接收缓存中内存块 n 的本地状态，接收缓存将使该块的本地副本失效（Invalidate-One-Line），或忽略该消息（忽略单行失效）。

为了说明 SYNCHX 规则在此处的应用，我们来仔细考察 broadcastX 方法的定义。其方法体包含一个全局同步模式的实例（图4）。如第 2 节所述，由于全局合流性质，我们可以假设其执行是原子的，即不会与它未生成的任何进程交错执行。对总线 sendRdX 方法的同步调用会生成对所有缓存（发起全局总线同步的缓存除外）的 receiveRdX 方法的异步调用（图 12）。按照通用的全局同步模式（图4），这些方法调用由一个 开始 和一个 结束 屏障进行同步。 receiveRdX 方法开头注释中的两个条件描述了两条可能的执行路径及其对应的转换规则INVALIDATE-ONE-LINE和 IGNORE-INVALIDATE-ONE-LINE。

如果条件 res==真 不成立，那么根据注释，没有转移规则对应于 broadcastX 方法的执行。在这种情况下，由 broadcastX方法（图 11）调用的总线同步失败，因为一级缓存中该地址的状态不再是共享状态（而 PRWR_2 规则要求该状态为共享状态）。因此，write(n) 指令本身的处理失败，并将通过对 run 方法的异步自调用再次处理。

我们以注释 run 方法的非正式说明作为结尾，注释为已移除==真： PRWR_3 方法的同步调用 移除_inv（第 22 行）。其条件描述了与以下转移规则对应的路径：

$$\frac{first(caid) = true \quad cid(caid) = c \quad status(M, n) \in \{inv, \bot\}}{(c \bullet \mathbf{write}(n); rst), (caid \bullet M \bullet dst) \rightarrow (c \bullet \mathbf{writeBI}(n); rst), (caid \bullet M[n \mapsto \bot] \bullet dst + \mathbf{fetch}(n))} \text{(PRWR}_3\text{)}$$

此规则处理核心尝试向地址为 n 的内存块写入数据的情况，该内存块要么处于无效状态，要么未在一级缓存中找到。此时核心将被阻塞，同时从

较低级别的缓存或主存中获取该内存块。另一方面，根据注释，没有转移规则对应于由条件取反所描述的执行路径。请注意，这涵盖了 getStatus 返回的状态（第 16 行）已发生变化的情况；也就是说，该内存块的状态不再是未定义或无效状态。与上述情况一样，run 方法会终止，但未能成功处理 write(n) 任务；该任务将在下一次异步调用 run 方法时重新评估。

在下一　节中，我们将展示如何通过模拟关系，对注释进行形式化验证　.

6 模拟关系

现在，我们将证明多核内存系统的 ABS 实现相对于多核 TSS 规范的正确性。首先我们注意到，第5节中描述的结构对应关系，仅关联 ABS 程序的类图（图 7）与多核 TSS 运行时配置的语法（图 6）。为了关联行为信息，我们定义抽象函数 α，它将每个稳定的全局 ABS 配置 G 映射到结构等价的配置 $\alpha(G)$，并且还在 ABS 类 Cache 实例的可观察进程与相应 TSS 缓存表示 Ca的一一映射dst指令之间建立映射，使得相关联的dst 指令的实际地址等于 ABS 进程对应形式参数的值。可观察的 ABS 进程是指由异步调用某个方法所产生的进程，该方法对应于一条dst 指令（如 fetch 等）。

下面给出主定理，说明 ABS 多核程序是多核 TSS 的一个正确实现，这是定理 2 的一个实例（请回忆，$\Rightarrow$ 表示稳定 ABS 配置之间的转移关系）：

定理 2. 设 G 为 *ABS* 多核模型的一个可达稳定全局配置。若 $G \Rightarrow G'$，则$\alpha(G) = \alpha(G')$ 或 $\alpha(G) \rightarrow \alpha(G')$。

定理 2 的证明. 由于 ABS 语义具有一般的合流性质，只需根据模拟关系 α，验证 run 方法的注释以及 Cache 类中与 dst 指令对应的方法的注释。下面详细分析 Cache 类。

我们首先验证 fetch 方法（图 13）的注释，并根据行号标识稳定点。fetch 方法涉及对辅助方法 broadcast（图 13）和 swap（图 14）的同步调用。broadcast 方法描述了全局同步模式（图4）的一个实例。总线的 sendRd 方法异步调用所有缓存（调用缓存除外）的 receiveRd 方法（见图 15），并使用屏障同步（同样参见图4、第 2 节）。swap 方法是测试并设置模式的一个实例，如图 2 所示。

令 $G \Rightarrow G'$ 描述一次 fetch 方法调用从一个稳定点到下一个稳定点（或终止）的执行，并令 $caid \bullet M \bullet$ dst $+$ fetch(n) 表示 $\alpha(G)$ 中与正在执行 fetch 方法的 G 中的缓存相对应的缓存，其中 n 表示正在执行的 fetch 方法调用的形式参数 n 的值。此外，令 $caid' \bullet M' \bullet$ dst′ 表示 $\alpha(G)$ 中与 $\alpha(G)$ 中的下一级缓存相对应的缓存（如果已定义）。如果存在这样的下一级缓存，则有 lid ($caid'$) $=$ lid ($caid$) $+ 1$，且 cid ($caid$) $=$ cid($caid'$)。下文中，我们用$\overline{M}$表示 $\alpha(G)$ 中的主存。对于 $G \Rightarrow G'$，我们有以下情形分析，它们描述了一次 fetch 方法调用从一个稳定点到下一个稳定点（或终止）的执行。

行 $1 \Rightarrow 5$。在这种情况下，$G \Rightarrow G'$ 涉及由缓存对象执行 fetch 方法，从方法开头（第 1 行）开始，直到对下一级缓存的 remove_inv 方法进行同步调用（第 5 行）（注意，因此 nextCache != 无成立）。由于这会使与 fetch 方法开头相关联的注释的路径条件无效，根据定义（参见第 3.1 节），不存在与 $G \Rightarrow G'$ 对应的（多核 TSS）转换规则。此次执行仅将此调用添加到下一级缓存的队列中，并且 α 抽象掉对 remove_inv 方法的调用，因此可知$\alpha(G') = \alpha(G)$。

行 $5 \Rightarrow 8$。在这种情况下，$G \Rightarrow G'$ 由从被调用的执行开始，经过后续移除_inv 方法的执行路径，经由**then**分支和后续**if**语句，直至获取 方法终止的路径组成（第 8 行）。根据注释 已移除==真： LC 未命中，此执行路径应对应于以下

```
1  单元 获取(地址 n){
2  // 下一级==无： LLC-Miss/Synch
3    分支选择(下一级) {
4      Just(下一级缓存) => {
5        布尔值 已移除 = 下一级缓存.移除_inv(n); // 已移除==真： LC 未命中
6        如果 (已移除){
7          nextCache!fetch(n);
8          this!fetchBl(n); }
9        否则 {Pair<地址,状态> 已选择 = select(cacheMemory, n);
10         Maybe<状态> s = nextCache.swap(n,selected);
11         //s!=无|& fst(selected)==n: LC-Hit2;
12         //s!=无|& fst(selected)!=n: LC-Hit1
13         如果 (s!= 无){如果 (fst(selected)!=n){
14           缓存内存 = 移除键(缓存内存,fst(selected)); }
15           缓存内存 = 放入(缓存内存, n,fromJust(s)); }
16         否则这个!fetch(n);} }
17       _ =>{ 此.广播(n);
18             此!fetchBl(n);} } }
19
20 单元 broadcast(地址 n){
21   等待 总线!锁();
22   总线.发送读请求(this, n);
23   bus.release(); }
```

图13。 带注释的获取方法。

对 LC 未命中 规则的后续应用：

$$\frac{lid(caid') = lid(caid) + 1 \quad cid(caid) = cid(caid') \quad status(M', n) \in \{inv, \bot\}}{(caid \bullet M \bullet \boxed{dst + \mathsf{fetch}(n)}),\ (caid' \bullet \boxed{M' \bullet dst'}) \rightarrow (caid \bullet M \bullet \boxed{dst + \mathsf{fetchBl}(n)}),\ (caid' \bullet \boxed{M'[n \mapsto \bot] \bullet dst' + \mathsf{fetch}(n)})} \text{(LC-Miss)}$$

此规则处理如下情况：缓存尝试从其下一级缓存获取内存块 n，但该块已失效或在缓存中不存在。于是，此缓存中的 fetch 方法将被挂起；获取操作会传播到下一级缓存，并且内存块 n 将从下一级缓存中移除。由于在这种情况下，remove inv 方法返回了布尔值“真”，我们可以根据其代码（图 11，第5节）静态推断：给定地址在下一级缓存中的初始状态为无或 Just(无效状态)。因此，在 $\alpha(G)$ 中，应用 LC-Miss 规则的条件已满足。此外，在这种情况下，我们可以静态推断，fetch 方法的执行由更新 dst $+\,\mathsf{fetchBl}(n)$ 和 $\mathrm{dst}' + \mathsf{fetch}(n)$ 模拟，而 remove_inv 方法由更新 $M'[n \mapsto \bot]$ 模拟。由此得出，应用 LC-Miss 规则后，$\alpha(G) \rightarrow \alpha(G')$。

行 $5 \Rightarrow 10$。*</style}*在这种情况下，由这样一条路径组成：从执行 *removeinv* 方法开始，经由后续 *if* 语句的 **else** 分支，到达对 swap 方法的（同步）调用（第10行）。由于在这种情况下，方法 remove_inv 返回了布尔值“False”，因此配置 G' 是通过初始化局部变量 selected 并将对 swap 方法的调用排入队列而从 G 得到的。我们暂且不考虑 ABS select 函数的定义；该函数会选择一个缓存行以供驱逐，从而为新获取的内存块腾出空间。我们简单地假设，ABS表达式 select(cacheMemory,n) 所表示的对中的第一个元素等于地址 $select(M, n)$，其第二个元素等于该地址的状态。[9] 根据模拟关系的定义，该关系忽略局部变量以及诸如 swap 方法之类的辅助方法调用，因此可得 $\alpha(G') = \alpha(G)$。

行 $10 \Rightarrow 15$。在此情况下，$G \Rightarrow G'$ 描述了从 swap 方法的执行经由后续**if**语句的**then**分支（因此 s != 无）到达 fetch 方法终止处（第 15 行）的两条路径之一。我们可以从 s != 无静态推断出 $M'(n) = s'$，其中 $s' =$ 共享状态 $\vee\, s' =$ 修改状态。此外，

[9] 在实际的 A BS 实现中，使用额外参数来表示缓存的最大容量，以检查 if 是否有空闲空间 可从下一级获取内存块 n。

```
Maybe<状态> swap(地址 n    _out, 对<地址,状态> n        _in) {
  Maybe<状态> tmp  =无;
  分支选择 (查找(缓存内存,n     _out)) {
    无 => 跳过;
    Just(无效状态) => 跳过;
    _ => {
      tmp = 查找(缓存内存,n        _out);
      缓存内存  =移除键(缓存内存,n                _out);
      如果(fst(n  _in)!=n    _out) {
        缓存内存  =放入(缓存内存, fst(n          _in), snd(m      无效状态)); } }
  返回tmp;}
```

图 14. 交换方法。

```
单元 receiveRd(地址 n,IBarrier 开始,IBarrier end) {
  // 查找(缓存内存,n))==Just(修改状态): FLUSH-ONE-LINE;
  // 查找(缓存内存,n))!=Just(修改状态): 忽略单行刷新
  等待 开始!同步();
  分支选择 (查找(缓存内存,n)) {
    Just(修改状态) => this!刷新(n);
    _ => 跳过;
  }
  结束!同步(); }
```

图 15。 带注释的 receiveRd 方法。

由于 G 可达，我们可以根据同步调用的语义推断，对于执行 fetch 方法的给定缓存对象，selected== select(cacheMemory,n) 在 G 中成立。最后，我们注意到，在 fetch 方法的第 13 行，fst(selected) ==n 成立，当且仅当在 swap 方法的第 9 行，fst(n_无效状态) == n_out 成立。根据 fst(selected) 的值，存在两种情况。

首先，令 fst(s elected) != n，也就是说，$select(M,\ n)\ \neq n$。由此可知，LC-HI T_1 rule

$$\frac{\begin{array}{c}(\text{LC-HIT}_1)\\ lid(caid') = lid(caid)+1 \quad cid(caid) = cid(caid')\\ select(M,n)=m \quad n\neq m \quad M(m)=s \quad M'(n)=s' \quad s'=sh\vee s'=mo\end{array}}{\begin{array}{c}(caid\bullet\ M\ \bullet dst+\textbf{fetch}(n)),\ (caid'\bullet\ M'\ \bullet dst') \rightarrow\\ (caid\bullet\ M[m\mapsto\bot,n\mapsto s']\ \bullet dst),\ (caid'\bullet\ M'[n\mapsto\bot,m\mapsto s]\ \bullet dst')\end{array}}$$

在 $\alpha(G)$ 中满足。规则 LC-Hit$_1$ addresses the case where a cache finds n，即待获取的块，在下一级中处于共享状态或已修改状态，且缓存中没有可用空间来放置 n。为了获取 n，该规则会在当前缓存中选择一个内存块 m，与下一级缓存中的 n 进行交换。根据注释 s != 无 & fst(selected) != n : LC-Hit$_1$, ，此规则应对应于由其条件标识的路径。由于 fst(n_in) != n_out，因此，ABS 中下一级缓存执行交换方法由更新 $M'[n\mapsto\bot,m\mapsto s]$ 模拟；另一方面，fetch 方法（第 14 行和第 15 行）中的两次赋值则由更新 $M[m\mapsto\bot,n\mapsto s']$模拟。

接下来，令 fst(selected)==n，也就是说，$select(M,n)\ ==n$。由此可得 LC-HIT$_2$规则的启用条件

$$\frac{\begin{array}{c}(\text{LC-HIT}_2)\\ lid(caid') = lid(caid)+1 \quad cid(caid) = cid(caid')\\ select(M,n)=n \quad M'(n)=s' \quad s'=sh\vee s'=mo\end{array}}{\begin{array}{c}(caid\bullet\ M\ \bullet dst+\textbf{fetch}(n)),\ (caid'\bullet\ M'\ \bullet dst') \rightarrow\\ (caid_i\bullet\ M[n\mapsto s']\ \bullet dst),\ (caid'\bullet\ M'[n\mapsto\bot]\ \bullet dst')\end{array}}$$

均已满足。此规则处理以下情况：缓存从其下一级缓存中找到 n，即待获取的内存块，且其状态为共享状态或修改状态，同时当前缓存中有可用空间可放置 n。根据

注释s! = 无 & fst(selected) ==n: LC-Hit$_2$, 此规则应对应于由其条件确定的路径。由于fst(n_in) == n_out，在此情况下，ABS 中下一级缓存执行swap 方法的过程由 $M'[n \mapsto \bot]$ update 模拟；另一方面，赋值操作（fetch 方法的第 15 行）的执行由 update $M[n \mapsto s']$模拟。

行 $10 \Rightarrow 16$。在这种情况下， $G \Rightarrow G'$ 描述了这样一条执行路径：它从 swap 方法返回后开始，经过后续 if 语句的 **else** 分支（因此 s == 无），最终到达 fetch 方法的终止。根据 swap 方法的代码以及条件 s == 无，我们可以静态推断，fetch 方法的形式参数 n 所指地址在下一级缓存中的状态未定义或无效。根据 swap 方法的注释，多核 TSS 中没有规则适用。由于此次 fetch 方法调用在异步自调用（传递相同地址）后终止，因此我们有 $\alpha(G) = \alpha(G')$。

行 $1 \Rightarrow 18$ 在这种情况下， $G \Rightarrow G'$ 涉及从方法开头开始执行fetch 方法，并在执行广播方法以及对 fetchBl 方法的异步自调用后终止。方法广播实现了全局同步模式的一个实例（图4）。对总线的发送读请求方法的同步调用会生成对所有缓存的receiveRd 方法的异步调用（图 12），但发起全局总线同步的缓存除外。根据注释LLC-Miss/Synch，此执行路径对应于全局同步规则

$$\frac{\overline{Ca} \xrightarrow{!Rd(n)} \overline{Ca'}}{\langle \overline{CR},\ \overline{Ca},\ M\rangle \to \langle \overline{CR},\ \overline{Ca'},\ M\rangle}\ (\text{Synch})$$

其中，前提是通过连续应用该规则生成的

$$\frac{Ca_1 \notin \overline{Ca} \quad \overline{Ca} \xrightarrow{!Rd(n)} \overline{Ca'} \quad Ca_1 \xrightarrow{?Rd(n)} Ca_2'}{\overline{Ca} \cup \{Ca_1\} \xrightarrow{!Rd(n)} \overline{Ca'} \cup \{Ca_2\}}\ (\text{Synch-Dist})$$

后一个规则本身由以下规则触发

$$\frac{last(caid) = true}{(caid \bullet M \bullet \ dst + \mathbf{fetch}(n)\ ) \xrightarrow{!Rd(n)} (caid \bullet M \bullet \ dst + \mathbf{fetchBl}(n)\ )}\ (\text{LLC-Miss})$$

and

$$\frac{\text{状态}(M,n) = \text{修改状态}}{caid \bullet M \bullet \ dst \xrightarrow{?Rd(n)} caid \bullet M \bullet \ dst + \mathbf{flush}(n)}\ (\text{Flush-One-Line}) \qquad \frac{\text{状态}(M,n) \neq \text{修改状态}}{caid \bullet M \bullet dst \xrightarrow{?Rd(n)} caid \bullet M \bullet \text{目标}}\ (\text{忽略单行刷新})$$

这些规则共同刻画了多核内存系统中获取内存块最新共享副本的广播机制。规则 LLC-Miss 对应于末级缓存需要获取内存块 n 的情况：它向所有其他缓存广播一条读消息 $!Rd(n)$。这是通过触发全局同步规则 Synch 和 Synch-Dist 实现的。前者确定广播读消息的核心 CR，后者则将消息 $!Rd(n)$ 传播到其他缓存。根据块 n 的状态，接收缓存要么将本地已修改副本刷新到主存（Flush-One-Line），要么忽略该消息（忽略单行刷新）。

显然，receiveRd 方法由上述规则 Flush-One-Line 和 Ignore-Flush-One-Line 模拟。由于我们可以假设（如第 2 节所述），广播 方法的执行仅由其生成的进程交错执行而成，因此很容易静态验证以下执行由上述规则模拟：**this**.broadcast(**this**,n);**this**!fetchBl(n) 。

剩下的是 fecthBl、fetchW 和 flush 方法的正确性（见图 16、17 和 18）。这可以采用与上述 fetch 方法的正确性证明类似的方式来确立。下面我们讨论这些正确性证明，省略那些易于核查的细节。

fetchBl 方法开头的注释条件标识了这样一条路径：该路径在调用 fetchW 方法后终止（第 23 行）。不难验证，这条路径由以下内容模拟：

rule

$$\frac{last(caid) = true \quad select(M,n) = n' \quad n' \neq n \quad status(M,n') = mo}{(caid \bullet M \bullet \boxed{dst + \textbf{fetchBl}(n)}) \rightarrow (caid \bullet M \bullet \boxed{dst + \textbf{flush}(n') + \textbf{fetchW}(n,n')})} \ (\textsc{FetchBl}_3)$$

处理末级缓存因 n 而被阻塞的情况，且 n 要放置的位置已被一个已修改的内存块占据。为了腾出 n的位置，需要通过 **刷新**指令将已修改的块刷新到主存中，随后获取操作将通过 **fetchW** 指令继续执行。根据定义，从 fetchBl 方法开头开始、并到达 getStatus 方法同步调用处的执行路径（第 7、17 和 25 行）均不对应任何规则。

注释 状态! = 无： LLC-Fetch-Unblock 标识出一条路径，该路径从执行下一级缓存的 getStatus 方法（第 7 行）开始，接着执行后续的 **else**- 分支和随后的 **if**-语句，直至终止。显然可以验证，该路径由以下规则模拟：

$$\frac{lid(caid') = lid(caid) + 1 \quad cid(caid) = cid(caid') \quad n \in dom(M')}{\begin{array}{c}(caid \bullet M \bullet \boxed{dst + \textbf{fetchBl}(n)}), (caid' \bullet M' \bullet dst') \rightarrow \\ (caid \bullet M \bullet \boxed{dst + \textbf{fetch}(n)}), (caid' \bullet M' \bullet dst')\end{array}} \ (\textsc{LC-Fetch-Unblock})$$

对应于这样的情况： n，即缓存所阻塞等待的内存块在下一级缓存中找到。该规则通过再次尝试获取 n 来解除缓存的阻塞（将 **fetchBl**(n) 替换为 **fetch**(n) ，在 dst中）。根据定义，没有规则对应于执行 **then**-分支后直至终止的路径。在这种情况下，执行 fetchBl 方法时会忙等待，直到下一级缓存中给定地址的状态变为已定义。

注释 $\textsc{FetchBl}_1$ 标识出一条路径，该路径从执行主存的 getStatus 方法以及更新 缓存内存 = put(cacheMemory,n,status) （第 17 行）开始，直至终止。显然可以验证，该路径由以下规则模拟：

$$\frac{last(caid) = true \quad select(M,n) = n \quad s = status(\overline{M},n)}{(caid \bullet \boxed{M \bullet dst + \textbf{fetchBl}(n)}), \overline{M} \rightarrow (caid \bullet \boxed{M'[n \mapsto s] \bullet dst}), \overline{M}} \ (\textsc{FetchBl}_1)$$

该规则会解除有空间可获取内存块的末级缓存的阻塞 n ，使其能够从主存获取内存块 M。类似地，注释$\overline{\textsc{FetchBl}_2}$ 标识了这样一条路径：它从执行主存的getStatus 方法（第 25 行）开始，经过后续更新removeKey(cacheMemory,selected_n) 和 cacheMemory =put(cacheMemory,n,status)，最终到达终止状态。很容易验证，该路径由规则模拟：

$$\frac{\text{末}(caid) = true \text{ 选择}(M,n) = n' \quad n' \neq n \text{ 状态}(M,n') \neq \text{修改状态 } s = \overline{\text{状态}(M,n)}}{(caid \bullet \boxed{M' \bullet dst + \textbf{fetchBl}(n)}), \overline{M} \rightarrow (caid \bullet \boxed{M'[n' \mapsto \bot, n \mapsto s] \bullet dst}), \overline{M}} \ (\text{fetchBl}_2)$$

这对应于末级缓存从主存获取内存块 n ，但用于放置 n 的位置已被一个未修改的块占用。随后，规则移除占用该位置的块，并将 n 放入缓存。

对于 fetchW 方法（图17），很容易验证，从 **await**语句开始直至终止的路径由该规则模拟。

$$\frac{last(caid) = true \quad status(M,n') \neq mo}{(caid \bullet M \bullet \boxed{dst + \textbf{fetchW}(n,n')}) \rightarrow (caid \bullet M \bullet \boxed{dst + \textbf{fetchBl}(n)})} \ (\textsc{FetchW})$$

该规则处理这样的情况：末级缓存由于 n' 是一个已修改块而无法用 n 替换 n' 时，可以再次尝试获取 n 。

```
单元 fetchBl(地址 n){
  // 下一级==无 &
  // fst(select(缓存内存, n)) !=n &
  // select(缓存内存, n)==对(        _,修改状态): 获取Bl3;
  分支选择（下一级）{
    Just(下一级缓存) => {
      Maybe<状态> status = 下一级缓存.getStatus(n);
      // 状态!  =无:  LC-FETCH-UNBLOCK
      如果 （状态 == 无）{
        此!  fetchBl(n); }
      否则 {
        此!fetch(n); }
      }
     _ => {
      对<地址,状态>已选择  =select(cacheMemory, n);
      如果 (fst(selected)==n){

       状态 status  =mainMemory.getStatus(n); // FETCHBL1
        缓存内存  = 放入(cacheMemory,n,status); }
      否则 {
        分支选择 (selected){
          对(selected   _n,修改状态)  =>{
            this!刷新(selected   _n);
            此!fetchW(n,已选择  _n); }
          对(selected   _n,_)  =>{
            状态 status  = 主存.getStatus(n); // 获取BL2
            缓存内存  =移除键(cacheMemory,selected        _n);
            缓存内存  = 放入(cacheMemory,n,status); }
        } } } } }
```

图 16. 带注释的 fetchBl 方法。

```
  单元 fetchW(地址 n,地址 n        _){
   等待 (查找默认值(缓存内存,n          _, 无效状态)!  =修改状态); // 获取
   此!fetchBl(n);
  }
```

图 17。 带注释的 fetchW 方法。

```
单元 刷新(地址 n) {
//查找(缓存内存,n)!  =修改状态:  刷新2;
  分支选择 (查找(缓存内存,n)) {
    Just(修改状态) => {
      主存.设置状态(n,共享状态); //刷新1
      缓存内存  = 放入(cacheMemory,n,共享状态); }
    _ => 跳过;
  } }
```

图18. 带注释的刷新方法。

由注释条件标识的路径 查找(缓存内存,n) != 修改状态： 刷新$_2$ 的 刷新方法（图18）显然由以下规则模拟

$$\frac{(\textsc{Flush}_2)\quad status(M,n) \neq mo}{(caid \bullet M \bullet \; dst + \mathbf{flush}(n)\;) \rightarrow (caid \bullet M \bullet \; dst\;)}$$

它仅在内存块未被修改时忽略刷新指令。根据此注释的定义，没有任何规则对应于通向方法同步调用的路径 setStatus ，即主存的该方法。根据注释FLUSH$_1$, 从主存的setStatus 方法执行到终止的路径，对应于规则

$$\frac{(\text{FLUSH}_1)\quad status(M,n) = mo}{(caid \bullet \ M \bullet dst + \textbf{flush}(n)\ ),\ \overline{M} \ \rightarrow (caid \bullet \ M[n \mapsto sh] \bullet dst\ ),\ \overline{M}[n \mapsto sh]}$$

它通过将内存块 n 的状态在缓存和主存中都更新为共享状态，从而刷新已修改的内存块。显然，根据模拟关系，此规则的条件在 $\alpha(G)$ 中成立。此外，更新 cacheMemory =put(cacheMemory,n,Sh) 由 $M[n \mapsto sh]$ 模拟，而更新 mainMemory.setStatus(n,Sh) 由 $M[n \mapsto sh]$ 模拟。

6.1 互模拟

我们简要讨论如何将定理 2 扩展为 ABS 多核程序的转移关系 $\Rightarrow$ 的传递自反闭包与多核 TSS 的转移关系 $\rightarrow$ 的传递自反闭包之间的互模拟。这样的互模拟关系进而允许我们根据多核 TSS 证明 ABS 多核程序的安全性和活性。以下定理指出，多核 TSS 可由 ABS程序模拟。

定理 3. 设 G 为 *ABS* 多核程序的一个可达稳定全局配置。若 $\alpha(G) \rightarrow cf$, 则存在一个稳定全局配置 G' ，使得 $\alpha(G') = cf$ 且 $G \Rightarrow^* G'$。

证明。 我们概述该定理的证明，其基于注释的正确性，而这已在定理 2 的证明中确立。然而，定理 3 的证明整体结构涉及对各个 TSS 规则的分析。所有这些规则均由一条 dst 指令触发。对于每一条此类指令，我们针对相应 ABS 方法的每个稳定点进行静态检查，以确定是否存在一条通往另一稳定点的路径（不一定是下一个稳定点），该路径的执行对应于 TSS规则应用。

作为该方案的一个示例，我们给出对该规则一次应用的分析

$$\frac{(\text{LC-MISS})\quad lid(caid') = lid(caid) + 1 \quad cid(caid) = cid(caid') \quad status(M', n) \in \{inv, \bot\}}{\begin{array}{c}(caid \bullet M \bullet \ dst + \textbf{fetch}(n)\ ),\ (caid' \bullet \ M' \bullet dst'\ ) \rightarrow \\ (caid \bullet M \bullet \ dst + \textbf{fetchBl}(n)\ ),\ (caid' \bullet \ M'[n \mapsto \bot] \bullet dst' + \textbf{fetch}(n)\ )\end{array}}$$

in $\alpha(G)$，由 dst 指令 **fetch**(n)触发。根据 α 的定义，存在一个 fetch 方法的进程实例（正在执行或已挂起），其形式参数的值为地址 n 。此外，该进程所属缓存对象的 nextCache 字段所指向的下一级缓存中，地址 n 的状态未定义或无效。我们对该进程的稳定点作如下直接分析。

对于初始稳定点以及与调用移除_inv 方法相关联的稳定点（获取方法的第 5 行，图 13），根据 α 的定义，存在一个计算 $G \Rightarrow^* G'$ ，在这两种情况下都包含从第 5 行到第 8 行的路径的执行。如定理 2 的证明中所述，该路径对应于 LC 未命中规则的应用。请注意，从第 1 行到第 5 行的路径执行对应于多核 TSS 中的一次静默迁移，

对于与 swap 方法调用相关联的稳定点（第10行），根据 α 的定义，存在一次计算 $G \Rightarrow^* G'$ ，其中包含如下路径的执行：该路径从 swap方法返回后，经过后续 **else**-分支的 **if**-语句（注意 s == 无），到 fetch方法终止，随后执行从 新生成的 fetch 方法进程实例的初始稳定点到第8行的路径。如上所述，后一条路径对应于 LC 未命中规则的一次应用。第一条路径的执行对应于多核 TSS 中的一次静默迁移（如定理 2 的证明中所述）。

7 相关工作

迁移规范与其在（高级）并行编程语言 [49]中的实现之间通常存在显著差距。迁移系统规范 [46] 能够简洁地形式化操作模型，且适用于证明，但直接实现此类规范很快就会导致效率极低的实现。Redex [24], 、重写逻辑 [43,42], 和𝕂 [47] 等可执行语义框架能够缩小这一差距，并已用于为 C [23] 和 Java [12]等复杂语言开发可执行的形式化模型。迁移系统规范与重写逻辑语义之间的关系已有研究 [50] ，但尚未提出基于标签匹配实现同步的一般解决方案。Bijo 等人将其多核内存模型 [8] 在重写逻辑系统 Maude [15] 中实现，并使用编排器进行标签匹配，但没有针对迁移系统规范提供正确性证明。不同的语义风格可以在同一框架内建模并建立关联；例如，已有研究使用重写逻辑 [22]，通过模拟关系考察 KLAIM 系统分布式实现的正确性。与这些语义研究相比，我们针对主动对象语言 ABS，开发了一种证明迁移系统规范的并行实现正确性的一般方法。我们的方法将这两种形式体系以一种新的方式结合起来，包括一种形式化方案，用于为 ABS 程序添加转移规则注释。这些注释为证明义务提供高层规范，以确立迁移系统规范与其 ABS 实现之间的模拟关系。

保正确编译和精化与实现的正确性证明相关，可确保程序的低级表示保留高级模型的性质。这方面的工作包括基于抽象状态机之间精化的 B 方法 [1],、向类型化汇编语言进行的保类型翻译 [45],，以及形式化验证的编译器 [36,37], 。后者证明了从 C 到汇编代码的编译器能够保留语义，但将共享变量并发留待 future 工作。与这些工作不同，我们的工作专门针对并行系统的正确性。

缓存一致性协议模拟工具可用于评估不同架构上的性能和效率（例如，GEMS [41] 和 gem5 [11]）。这些工具通过运行以低级内存读写指令编写的基准测试程序，评估缓存命中/未命中率和响应时间等指标。Graphite [44]和 Sniper [14] 等先进模拟器会在分布式集群上运行程序，以模拟数千个核心组成的多核架构上的执行。与我们的工作不同，这些模拟器并非基于形式语义和正确性证明。我们的工作与这些模拟器互为补充，支持从程序员视角可执行地探索设计选择，而非从硬件设计角度进行探索。与针对多核架构上并发程序的最坏情况响应时间分析 [38], 相比，我们关注的是底层数据移动，而非响应时间。

8 结论

本文介绍了一种方法论，用于证明主动对象语言 ABS 中高层迁移系统规范的并行实现的正确性。该证明方法通过建立描述 ABS 程序语义的转移系统与规范所描述的转移系统之间的模拟关系来实现。该证明方法利用 ABS 语义的一般全局合流性质，从而能够抽象掉并行进程的交错执行，并在模拟证明中聚焦于顺序代码的静态分析。我们方法论中一个颇具前景的进一步形式化及工具支持的自动化方向，是通过对顺序 ABS 代码进行符号执行，建立 ABS 程序与其规范之间的模拟关系。

并行执行中经常会引发的一个问题是公平性：在不同并行组件（此处为模拟核心）之间分配计算资源时，变动程度有多大。并行执行的公平性会影响仿真对预期（或理想化）众核硬件的近似准确性。为了确保仿真的公平性，我们在 ABS 中使用部署组件 [31] 。

一个部署组件(DC) 是一种 ABS 执行位置，创建时会配备一定数量的虚拟资源（例如执行速度、内存使用量、network 带宽），这些资源由部署在其中的对象共享。任何带注释语句 [Cost: x] S 都会使其 DC 的资源减少 x，然后完成执行；如果当前剩余资源不足，该语句的计算就会停滞。该语句 S 可以在全局符号时间的下一次推进时继续执行，此时所有 DC 的资源都会得到更新；当其 Cost 降至零时，执行最终完成。

我们利用 ABS 的资源建模，为系统的模拟核心分配执行速度相同（公平）的虚拟资源。每个 Core 对象都部署在单独的 DC 上，并固定设置为 Speed(1)

资源。每条指令的处理成本都相同 [Cost: 1] ——这是一种泛化，因为常见的处理器架构以不同速度执行不同指令（每条指令所需的周期数）；例如，JUMP 比 LOAD 更快。其结果是，所有核心在全局符号时钟的每个时间间隔内最多只能执行一条指令，因此不会有任何核心在处理自身指令时过度超前——当使用拥有 M 个核心的物理机器并行模拟 N 个核心时，就会出现这一问题，其中 N 远大于 M。

我们计划进一步扩展这一带部署组件的 ABS 多核模型，以模拟（面向对象）程序在多核架构上的执行。首先，我们将为抽象内存模型扩展数据。特别是，将内存位置的地址本身作为数据，便可对动态生成的对象结构采用不同的数据布局进行建模和模拟。

参考文献

1. Jean-Raymond Abrial. *Event-B 建模——系统与软件工程*. 剑桥大学出版社，2010年。2. Elvira Albert、Puri Arenas、Antonio Flores-Montoya、Samir Genaim、Miguel Gómez-Zamalloa、Enrique Martin- Martin、German Puebla 和 Guillermo Román-Díez。SACO：并发对象静态分析器。载于 Erika Ábrahám 和 Klaus Havelund 编，第*20*届国际系统构造与分析工具和算法会议论文集（*TACAS 2014*），《计算机科学讲义》第8413卷，第562–567页。施普林格，2014年。3. Elvira Albert、Frank S. de Boer、Reiner Hähnle、Einar Broch Johnsen、Rudolf Schlatte、Silvia Lizeth Tapia Tarifa 和 Peter Y. H. Wong。云架构资源管理的形式化建模与分析：使用实时 ABS 的工业案例研究。《面向服务的计算与应用》，8(4):323–339，2014年。4. Elvira Albert、Miguel Gómez-Zamalloa 和 Miguel Isabel。SYCO：并发对象系统化测试工具。载于 Ayal Zaks 和 Manuel V. Hermenegildo 编，第*25*届国际编译器构造会议论文集（*CC 2016*），第269–270页。美国计算机协会，2016年。5. Gregory R. Andrews. *多线程、并行与分布式编程基础*. Addison Wesley，2000年。6. Nikolaos Bezirgiannis、Frank S. de Boer 和 Stijn de Gouw。云服务的人在回路仿真。载于 Flavio De Paoli、Stefan Schulte 和 Einar Broch Johnsen 编，第*6*届 *IFIP WG 2.14* 欧洲面向服务与云计算会议论文集（*ESOCC 2017*），《计算机科学讲义》第10465卷，第143–158页。施普林格，2017年。7. Nikolaos Bezirgiannis、Frank S. de Boer、Einar Broch Johnsen、Ka I Pun 和 Silvia Lizeth Tapia Tarifa。使用主动对象实现结构化操作语义：多核内存系统案例研究。载于 Reiner Hähnle 和 Wil M. P. van der Aalst 编，第*22*届国际软件工程基础方法会议论文集（*FASE 2019*），《计算机科学讲义》第11424卷，第332–350页。施普林格，2019年。8. Shiji Bijo、Einar Broch Johnsen、Ka I Pun 和 S. Lizeth Tapia Tarifa。用于缓存一致性多核架构的 Maude 框架。载于第*11*届国际重写逻辑及其应用研讨会论文集（*WRLA 2016*），《计算机科学讲义》第9942卷，第47–63页。施普林格，2016年。9. Shiji Bijo、Einar Broch Johnsen、Ka I Pun 和 S. Lizeth Tapia Tarifa。具有多级缓存的多核架构并行执行形式模型。载于第*14*届国际组件软件形式化方面会议论文集（*FACS 2017*），《计算机科学讲义》第10487卷，第58–77页。施普林格，2017年。10. Shiji Bijo、Einar Broch Johnsen、Ka I Pun 和 Silvia Lizeth Tapia Tarifa。具有多级缓存的多核架构数据访问形式模型。《计算机编程科学》，179:24–53，2019年。11. Nathan Binkert、Bradford Beckmann、Gabriel Black、Steven K. Reinhardt、Ali Saidi、Arkaprava Basu、Joel Hestness、Derek R. Hower、Tushar Krishna、Somayeh Sardashti、Rathijit Sen、Korey Sewell、Muhammad Shoaib、Nilay Vaish、Mark D. Hill 和 David A. Wood。gem5 模拟器。*SIGARCH《计算机体系结构新闻》*，39(2):1–7，2011年。12. Denis Bogdanas 和 Grigore Rosu。K-Java：Java 的完整语义。载于 Sriram K. Rajamani 和 David Walker 编，第*42*届年度 *ACM SIGPLAN-SIGACT* 编程语言原理研讨会论文集（*POPL 2015*），第445–456页。美国计算机协会，2015年。13. Roland N. Bol 和 Jan Friso Groote。迁移系统规范中否定前提的含义。《*ACM*学报》，43(5):863–914，1996年。14. Trevor E. Carlson、Wim Heirman 和 Lieven Eeckhout。Sniper：探索抽象级别，以实现可扩展且准确的并行多核仿真。载于国际高性能计算、网络、存储与分析会议论文集（*SC*），第52:1–52:12页。美国计算机协会，2011年。15. Manuel Clavel、Francisco Durán、Steven Eker、Patrick Lincoln、Narciso Martí-Oliet、José Meseguer 和 Carolyn L. Talcott 编。*Maude* 全面介绍——高性能逻辑框架：如何用重写逻辑规范、编程和验证系统，《计算机科学讲义》第4350卷。施普林格，2007年。16. David E. Culler、Anoop Gupta 和 Jaswinder Pal Singh。*并行计算机体系结构：硬件/软件方法*. Morgan Kaufmann 出版社，第一版，1997年。

17. Ferruccio Damiani、Reiner Hähnle、Eduard Kamburjan 和 Michael Lienhardt。增量与特征的统一形式化编程模型。载于 Marieke Huisman 和 Julia Rubin 编，第*20*届国际软件工程基础方法会议论文集（*FASE 2017*），《计算机科学讲义》第10202卷，第424–441页。施普林格，2017年。18. Frank de Boer、Vlad Serbanescu、Reiner Hähnle、Ludovic Henrio、Justine Rochas、Crystal Chang Din、Einar Broch Johnsen、Marjan Sirjani、Ehsan Khamespanah、Kiko Fernandez-Reyes 和 Albert Mingkun Yang。主动对象语言综述。《*ACM*计算综述》，50(5):76:1–76:39，2017年10月。19. Stijn de Gouw、Frank S. de Boer、Richard Bubel、Reiner Hähnle、Jurriaan Rot 和 Dominic Steinhöfel。验证 OpenJDK 泛型集合的排序方法。《自动推理杂志》，62(1):93–126，2019年。20. Stijn de Gouw、Jacopo Mauro、Behrooz Nobakht 和 Gianluigi Zavattaro。ABS 中的声明式弹性。载于 Marco Aiello、Einar Broch Johnsen、Schahram Dustdar 和 Ilche Georgievski 编，第*5*届 *IFIP WG 2.14* 欧洲面向服务与云计算会议论文集（*ESOCC 2016*），《计算机科学讲义》第9846卷，第118–134页。施普林格，2016年。21. Crystal Chang Din、Richard Bubel 和 Reiner Hähnle。KeY-ABS：并发建模语言 ABS 的演绎验证工具。载于*CADE*，《计算机科学讲义》第9195卷，第517–526页。施普林格，2015年。22. Jonas Eckhardt、Tobias Mühlbauer、José Meseguer 和 Martin Wirsing。Maude 中 KLAIM 模型的语义、分布式实现与形式化分析。《计算机编程科学》，99:24–74，2015年。23. Chucky Ellison 和 Grigore Rosu。可执行的 C 语言形式语义及其应用。载于 John Field 和 Michael Hicks 编，第*39*届 *ACM SIGPLAN-SIGACT* 编程语言原理研讨会论文集（*POPL 2012*），第533–544页。美国计算机协会，2012年。24. Matthias Felleisen、Robert Bruce Findler 和 Matthew Flatt。使用 *PLT Redex* 进行语义工程. 麻省理工学院出版社，2009年。25. Rob van Glabbeek。论迁移系统规范的含义。载于 Jorge A. Pérez 和 Jurriaan Rot 编，第*26*届并发表达性国际研讨会与第*16*届结构化操作语义研讨会联合论文集（*EXPRESS/SOS 2019*），《EPTCS》第300卷，第69–85页，2019年。26. Jan Friso Groote 和 Frits W. Vaandrager。结构化操作语义与作为同余的互模拟。《信息与计算》，100(2):202–260，1992年。27. Reiner Hähnle。抽象行为规范语言：教程式介绍。载于*FMCO 2012*，《计算机科学讲义》第7866卷，第1–37页。施普林格，2012年。28. Carl Hewitt、Peter Bishop 和 Richard Steiger。面向人工智能的通用模块化 ACTOR 形式体系。载于第*3*届国际人工智能联合会议论文集，IJCAI'73，第235–245页，美国加利福尼亚州旧金山，1973年。Morgan Kaufmann 出版社。29. Einar Broch Johnsen、Reiner Hähnle、Jan Schäfer、Rudolf Schlatte 和 Martin Steffen。ABS：抽象行为规范的核心语言。载于 Bernhard Aichernig、Frank S. de Boer 和 Marcello M. Bonsangue 编，第*9*届组件与对象形式化方法国际研讨会论文集（*FMCO 2010*），《计算机科学讲义》第6957卷，第142–164页。施普林格，2011年。30. Einar Broch Johnsen、Jia-Chun Lin 和 Ingrid Chieh Yu。使用基于模型的预测比较 AWS 部署。载于*ISoLA*（*2*），《计算机科学讲义》第9953卷，第482–496页，2016年。31. Einar Broch Johnsen、Rudolf Schlatte 和 S. ̃Lizeth Tapia Tarifa。在定时面向对象模型中整合部署架构与资源消耗。《逻辑与代数编程方法杂志》，84(1):67–91，2015年。32. Eduard Kamburjan。使用条件同步检测形式化系统模型中的死锁。《欧洲软件科学技术协会电子通讯》，76，2018年。33. Eduard Kamburjan、Reiner Hähnle 和 Sebastian Schön。使用主动对象对铁路运营进行形式化建模与分析。《计算机编程科学》，166:167–193，2018年。34. Eduard Kamburjan、Marco Scaletta 和 Nils Rollshausen。Crowbar：用于主动对象演绎验证的行为符号执行。《*CoRR*》，abs/2102.10127，2021年。35. Eduard Kamburjan 和 Jonas Stromberg。形式化系统模型验证的工具支持：交互式可视化与需求可追踪性。载于*F-IDE@FM*，《EPTCS》第310卷，第70–85页，2019年。36. Xavier Leroy。现实编译器的形式化验证。《*ACM*通讯》，52(7):107–115，2009年。37. Xavier Leroy。经过形式化验证的编译器后端。《自动推理杂志》，43(4):363–446，2009年。38. Yan Li、Vivy Suhendra、Yun Liang、Tulika Mitra 和 Abhik Roychoudhury。共享缓存多核处理器上运行的并发程序的时序分析。载于 Theodore P. Baker 编，第*30*届 *IEEE* 实时系统研讨会论文集（*RTSS 2009*），第57–67页。IEEE计算机学会，2009年。39. Jia-Chun Lin、Jacopo Mauro、Thomas Brox Røst 和 Ingrid Chieh Yu。云应用的基于模型的可扩展性优化方法论。载于第*7*届云与服务计算国际研讨会论文集（SC^2 *2017*），第163–170页。IEEE计算机学会，2017年。40. Jia-Chun Lin、Ingrid Chieh Yu、Einar Broch Johnsen 和 Ming-Chang Lee。ABS-YARN：用于建模 Hadoop YARN 集群的形式化框架。载于 Perdita Stevens 和 Andrzej Wasowski 编，第*19*届国际软件工程基础方法会议论文集（*FASE 2016*），《计算机科学讲义》第9633卷，第49–65页。施普林格，2016年。

41. Milo M. K. Martin、Daniel J. Sorin、Bradford M. Beckmann、Michael R. Marty、Min Xu、Alaa R. Alameldeen、Kevin E. Moore、Mark D. Hill 和 David A. Wood。Multifacet 的通用执行驱动多处理器模拟器（GEMS）工具集。*SIGARCH*《计算机体系结构新闻》，33(4):92–99，2005年。42. José Meseguer 和 Grigore Rosu。重写逻辑语义项目。理论计算机科学，373(3):213–237，2007年。43. José Meseguer 和 Grigore Rosu。重写逻辑语义项目：进展报告。信息与计算，231 69，2013年 38: – 。44. Jason E. Miller、Harshad Kasture、George Kurian、Charles Gruenwald、Nathan Beckmann、Christopher Celio、Jonathan Eastep 和 Anant Agarwal。Graphite：面向多核处理器的分布式并行模拟器。载于第*16*届国际高性能计算机体系结构研讨会（*HPCA*）论文集，第1–12页。IEEE计算机协会，2010年。45. J. Gregory Morrisett、David Walker、Karl Crary 和 Neal Glew。从系统F到类型化汇编语言。*ACM*程序设计语言与系统汇刊，21(3):527–568，1999年。46. Gordon D. Plotkin。操作语义的结构化方法。《逻辑与代数编程期刊》，60-61:17–139，2004年。47. Grigore Rosu。 $\mathbb{K}$：面向编程语言和形式化分析工具的语义框架。载于《可信软件系统工程》，第186–206页。IOS出版社，2017年。48. Rudolf Schlatte、Einar Broch Johnsen、Eduard Kamburjan 和 Silvia Lizeth Tapia Tarifa。资源敏感型参与者的建模与分析：教程式介绍。载于*COORDINATION 2021*，《计算机科学讲义》第12717卷，第3–19页。施普林格，2021年。49. Rudolf Schlatte、Einar Broch Johnsen、Jacopo Mauro、Silvia Lizeth Tapia Tarifa 和 Ingrid Chieh Yu。释放猛兽：形式化方法遇上真实世界数据。载于协调至关重要——庆祝 *Farhad Arbab* 毕生科学成就的论文集，《计算机科学讲义》第10865卷，第107–121页。施普林格，2018年。50. Traian-Florin Serbanuta、Grigore Rosu 和 José Meseguer。操作语义的重写逻辑方法。信息与计算，207(2):305–340，2009年。51. Yan Solihin。《并行多核体系结构基础》。Chapman & Hall/CRC，第1版，2015年。52. Daniel J. Sorin、Mark D. Hill 和 David A. Wood。《内存一致性与缓存一致性入门》。Morgan & Claypool出版社，第1版，2011年。53. Gianluca Turin、Andrea Borgarelli、Simone Donetti、Einar Broch Johnsen、Silvia Lizeth Tapia Tarifa 和 Ferruccio Damiani。Kubernetes容器框架的形式化模型。载于编辑 Tiziana Margaria 和 Bernhard Steffen 的第*9*届国际形式化方法应用研讨会（*ISoLA 2020*）论文集，《计算机科学讲义》第12476卷，第558–577页。施普林格，2020年。54. Lars Tveito、Einar Broch Johnsen 和 Rudolf Schlatte。通过局部控制实现分布式主动对象的全局可复现性。载于编辑 Heike Wehrheim 和 Jordi Cabot 的第*23*届国际软件工程基础方法会议（*FASE 2020*）论文集，《计算机科学讲义》第12076卷，第140–160页。施普林格，2020年。

多核 TSS

多核 TSS 由核心、缓存和全局同步各自独立的 TSS 构成。一般来说，我们假设描述各个核心和缓存行为的无标记转移是在配置 cf 中应用的，因此此处省略显而易见的上下文规则。另一方面，对于带标记转移，我们引入显式同步规则，将其提升到特定上下文中。

核心转移规则

图19展示了　　基本核心指令的转移规则 **读取**(r)、**ReadBl**(r)、**写入**(r)以及 **WriteBl**$(r$　　$)$.

缓存转移规则

这些规则进一步按照各个rst 指令的独立 TSS 进行组织（图 20、21、22 和 23）。

全局同步转移规则

这些规则进一步组织为带标记转移的 TSS（图24），以及用于匹配这些带标记转移的规则的 TSS（图25）。

$$\frac{first(caid) = true \quad cid(caid) = c \quad status(M,\ n) \in \{sh, mo\}}{(c \bullet \quad) \ (caid \bullet M \bullet dst) \rightarrow (c \bullet \quad r \quad) \ (caid \bullet M \bullet dst)} \ (\text{PRRD}_1)$$

$$\frac{first(caid) = true \quad cid(caid) = c \quad status(M,\ n) \in \{inv, \bot\}}{(c \bullet \quad), (caid \bullet \quad M \bullet dst\ ) \rightarrow (c \bullet \ \textbf{readBl}(n); rst\ ), (caid \bullet \ M[n \mapsto \bot] \bullet dst + \textbf{获取}(n)\ )} \ (\text{PRRD}_2)$$

$$\frac{first(caid) = true \quad cid(caid) = c \quad n \in dom(M)}{(c \bullet \quad), (caid \bullet M \bullet dst) \rightarrow (c \bullet \quad , (caid \bullet M \bullet dst)} \ (\text{PRRD}_3)$$

$$\frac{first(caid) = true \quad cid(caid) = c \quad status(M,\ n) = mo}{(c \bullet \quad) \ (caid \bullet M \bullet dst) \rightarrow (c \bullet \quad r \quad) \ (caid \bullet M \bullet dst)} \ (\text{PRWR}_1)$$

$$\frac{first(caid) = true \quad cid(caid) = c \quad status(M,\ n) = sh}{(c \bullet \quad) \ (caid \bullet \quad M \quad \bullet dst) \xrightarrow{!RdX(n)} (c \bullet \ rst\ ), (caid \bullet \ M[n \mapsto mo] \ \bullet dst)} \ (\text{PRWR}_2)$$

$$\frac{first(caid) = true \quad cid(caid) = c \quad status(M,\ n) \in \{inv, \bot\}}{(c \bullet \quad) \ (caid \bullet \quad M \bullet dst\ ) \rightarrow (c \bullet \quad), (caid \bullet \quad M[n \mapsto \bot] \bullet dst + \textbf{获取}(n)\ )} \ (\text{PRWR}_3)$$

$$\frac{first(caid) = true \quad cid(caid) = c \quad n \in dom(M)}{(c \bullet \quad), \ (caid \bullet M \bullet dst) \rightarrow (c \bullet \textbf{V} \quad) \ (caid \bullet M \bullet dst)} \ (\text{PRWR}_4)$$

图 19. 以下操作的转换规则：**读取**(r)，**ReadBl**(r)，**写入**(r)和**WriteBl**(r)。

B 多核 ABS

本节汇总了本文讨论过的 ABS 模型中的所有方法，以全面展示多核内存系统的 ABS 实现。

$$\frac{\begin{array}{c}(\text{LC-Hit}_1)\\ lid(caid') = lid(caid) + 1 \quad cid(caid) = cid(caid')\\ select(M,n) = m \quad n \neq m \quad M(m) = s \quad M'(n) = s' \quad s' = sh \vee s' = mo\end{array}}{\begin{array}{c}(caid \bullet M \bullet dst + \textsf{fetch}(n)),\ (caid' \bullet M' \bullet dst') \rightarrow\\ (caid \bullet M[m \mapsto \bot, n \mapsto s'] \bullet dst),\ (caid' \bullet M'[n \mapsto \bot, m \mapsto s] \bullet dst')\end{array}}$$

$$\frac{\begin{array}{c}(\text{LC-Hit}_2)\\ lid(caid') = lid(caid) + 1 \quad cid(caid) = cid(caid')\\ select(M,n) = n \quad M'(n) = s' \quad s' = sh \vee s' = mo\end{array}}{\begin{array}{c}(caid \bullet M \bullet dst + \textsf{fetch}(n)),\ (caid' \bullet M' \bullet dst') \rightarrow\\ (caid_i \bullet M[n \mapsto s'] \bullet dst),\ (caid' \bullet M'[n \mapsto \bot] \bullet dst')\end{array}}$$

$$\frac{\begin{array}{c}(\text{LC-Miss})\\ lid(caid') = lid(caid) + 1 \quad cid(caid) = cid(caid') \quad status(M',n) \in \{inv, \bot\}\end{array}}{\begin{array}{c}(caid \bullet M \bullet dst + \textsf{fetch}(n)),\ (caid' \bullet M' \bullet dst') \rightarrow\\ (caid \bullet M \bullet dst + \textsf{fetchBl}(n)),\ (caid' \bullet M'[n \mapsto \bot] \bullet dst' + \textsf{fetch}(n))\end{array}}$$

$$\frac{\begin{array}{c}(\text{LLC-Miss})\\ last(caid) = true\end{array}}{(caid \bullet M \bullet dst + \textsf{fetch}(n)) \xrightarrow{!Rd(n)} (caid \bullet M \bullet dst + \textsf{fetchBl}(n))}$$

图20. 用于 **获取**(n) 的转换规则。

$$\frac{\begin{array}{c}(\text{FetchBl}_1)\\ last(caid) = true \quad select(M,n) = n \quad s = status(\overline{M}, n)\end{array}}{(caid \bullet M \bullet dst + \textsf{fetchBl}(n)),\ \overline{M} \rightarrow (caid \bullet M'[n \mapsto s] \bullet dst),\ \overline{M}}$$

$$\frac{\begin{array}{c}(\text{FetchBl}_2)\\ last(caid) = true \quad select(M,n) = n' \quad n' \neq n \quad status(M,n') \neq mo \quad s = status(\overline{M}, n)\end{array}}{(caid \bullet M' \bullet dst + \textsf{fetchBl}(n)),\ \overline{M} \rightarrow (caid \bullet M'[n' \mapsto \bot, n \mapsto s] \bullet dst),\ \overline{M}}$$

$$\frac{\begin{array}{c}(\text{FetchBl}_3)\\ last(caid) = true \quad select(M,n) = n' \quad n' \neq n \quad status(M,n') = mo\end{array}}{(caid \bullet M \bullet dst + \textsf{fetchBl}(n)) \rightarrow (caid \bullet M \bullet dst + \textsf{flush}(n') + \textsf{fetchW}(n,n'))}$$

$$\frac{\begin{array}{c}(\text{LC-Fetch-Unblock})\\ lid(caid') = lid(caid) + 1 \quad cid(caid) = cid(caid') \quad n \in dom(M')\end{array}}{\begin{array}{c}(caid \bullet M \bullet dst + \textsf{fetchBl}(n)),\ (caid' \bullet M' \bullet dst') \rightarrow\\ (caid \bullet M \bullet dst + \textsf{fetch}(n)),\ (caid' \bullet M' \bullet dst')\end{array}}$$

图21. 用于 **fetchBl**(n) 的转换规则。

$$\frac{(\text{获取W}) \quad last(caid) = true \quad status(M,\ n') \neq mo}{(caid \bullet M \bullet \ dst + \textsf{fetchW}(n, n')\ ) \rightarrow (caid \bullet M \bullet \ dst + \textsf{fetchBl}(n)\ )}$$

图 22. fetchW 的转移规则 $\textsf{fetchW}(n,\ n')$。

$$\frac{(\text{刷新}_1) \quad status(M,\ n) = mo}{(caid \bullet \ ,\ \ \overline{M} \rightarrow (caid \bullet \ M[n \mapsto sh] \bullet dst\ ),\ \ \overline{M}[n \mapsto sh]}$$

$$\frac{(\text{刷新}_2) \quad status(M,\ n) \neq mo}{(caid \bullet M \bullet \ ) \rightarrow (caid \bullet M \bullet \ )}$$

图 23。 刷新的转换规则 $\textsf{刷新}(n)$。

$$\frac{(\textsc{Invalidate-One-Line}) \quad status(M,\ n) = sh}{caid \bullet \ dst \xrightarrow{?\ RdX(n)} caid \bullet \ dst}$$

$$\frac{(\text{忽略单行失效}) \quad status(M,\ n) \in \{inv, \bot\}}{caid \bullet M \bullet dst \xrightarrow{?RdX(n)} caid \bullet M \bullet dst}$$

$$\frac{(\textsc{Flush-One-Line}) \quad status(M,\ n) = mo}{caid \bullet M \bullet \ \xrightarrow{?Rd(n)} caid \bullet M \bullet \ }$$

$$\frac{(\text{忽略单行刷新}) \quad status(M,\ n) \neq mo}{caid \bullet M \bullet dst \xrightarrow{?Rd(n)} caid \bullet M \bullet dst}$$

图 24. 标记输入转换。

$$\frac{(\textsc{Synch-Dist}) \quad Ca_1 \notin \overline{Ca} \quad \sout{Ca} \xrightarrow{!Rd(n)} \sout{Ca'} \quad Ca_1 \xrightarrow{?Rd(n)} Ca_2'}{\sout{Ca} \cup \{Ca_1\} \xrightarrow{!Rd(n)} \sout{Ca'} \cup \{Ca_2\}}$$

$$\frac{(\text{同步}) \quad \sout{Ca} \xrightarrow{!Rd(n)} \sout{Ca'}}{\langle \overline{CR},\ \overline{Ca},\ M\rangle \rightarrow \langle \overline{CR},\ \overline{Ca'},\ M\rangle}$$

$$\frac{(\textsc{SynchX}) \quad Ca_1 \notin \overline{Ca} \quad CR, \overline{Ca} \xrightarrow{!RdX(n)} CR', \overline{Ca'} \quad Ca_1 \xrightarrow{?RdX(n)} Ca_2}{CR, \overline{Ca} \cup \{Ca_1\} \xrightarrow{!RdX(n)} CR', \overline{Ca'} \cup \{Ca_2\}}$$

$$\frac{(\textsc{SynchX}) \quad CR \notin \overline{CR_1} \quad CR, \sout{Ca} \xrightarrow{!RdX(n)} CR', \overline{Ca'}}{\langle \overline{CR_1} \cup \{CR\},\ \overline{Ca},\ M\rangle \rightarrow \langle \overline{CR_1} \cup \{CR'\},\ \overline{Ca'},\ M[n \mapsto inv]\rangle}$$

图 25。 全局同步/广播的转换规则。

```
单元 运行 {
  如果当前任务!=空值 {
  分支选择(当前任务) {
     Cons(rst, 剩余部分) =>
     分支选择 (rst){
        读取(n) =>{
            已移除 = 一级缓存.移除 _inv(n); // 已移除==真: PrRd2; 已移除==False: PrRd1
            if (已移除){
               一级缓存!获取(n);
               当前任务 = Cons(ReadBl(n),剩余部分); }
             否则 {当前任务 = 剩余部分; } }
         ReadBl(n) =>{
           状态 = 一级缓存.getStatus(n); // 状态! =无: PrRd3
          如果 (状态 != 无) 当前任务 = Cons(读取(n),剩余部分); }
        Write(n) =>{
          状态 = 一级缓存.getStatus(n); // 状态==Just(修改状态): PrWr1
          分支选择 (状态) {
            Just(修改状态) => {当前任务 = 剩余部分; }
            Just(共享状态) =>{
               布尔值 res = 一级缓存.广播X(n); //res==真: PrWr2/SynchX
               如果 (res) {当前任务 = 剩余部分; } }
            _ => { 布尔值 已移除 =一级缓存.移除 _inv(n); // 已移除==真: PrWr3
                  如果 (已移除){l1!获取(n); 当前任务 = Cons(WriteBl(n),剩余部分); } } } }
        WriteBl(n) =>{
          Maybe<状态> 状态 =l1.getStatus(n); // 状态! =无: PrWr4
          如果 (状态 != 无)
             当前任务 = Cons(Write(n),剩余部分); }
     }
  这个！运行(); }
}
```

图 26。 带注释的 run 方法。

```
Maybe<状态> getStatus(地址 n) { 返回 查找(缓存内存,n); }

布尔值 移除_inv(地址 n){
  布尔值 answer =False;
  分支选择 (查找(缓存内存,n)) {
    无 => { answer = 真; }
    Just(无效状态) =>{ 缓存内存 = 移除键(缓存内存,n); answer = 真; }
    _ => 跳过;}
  返回 answer;}

布尔值 广播X(地址 n) {
  布尔值 res = 假;
  等待 总线!锁(); //(查找(缓存内存,n) ==Just(共享状态)): PrWr2/同步
  如果 (查找(缓存内存,n) ==Just(共享状态)){
    总线.sendRdX(此, n);
    缓存内存 =放入(缓存内存,n,修改状态);
    res = 真; }
    总线.释放();
  返回res;}
```

图 27. 类 Cache 的方法 getStatus、remove_inv 和 broadcastX。

```
单元 receiveRdX(地址 n,IBarrier start,IBarrier end) {
  // 查找(cacheMemory,n))==Just(共享状态): Invalidate-One-Line;
  // 查找(cacheMemory,n))!=Just(共享状态): 忽略单行失效
  等待start!同步();
  分支选择 (查找(缓存内存,n)) {
    Just(共享状态) => {缓存内存 = 放入(缓存内存,n,无效状态); }
    _ => 跳过;
  }
  结束.同步(); }
```

图 28. 带注释的 receiveRdX 方法。

```
单元 获取(地址 n){
// 下一级==无： LLC-Miss/Synch
  分支选择(下一级) {
    Just(下一级缓存) => {
      布尔值 已移除 = 下一级缓存.移除_inv(n); // 已移除==真: LC 未命中
      如果 (已移除){
        nextCache!fetch(n);
        this!fetchBl(n); }
      否则 {Pair<地址,状态> 已选 = select(缓存内存, n);
        Maybe<状态> s = nextCache.swap(n,selected);
        // s！=无|& fst(selected)==n： LC-Hit2;
        // s！=无|& fst(selected)！=n： LC-Hit1
        如果 (s!= 无){如果 (fst(selected)！=n){
          缓存内存 = 移除键(缓存内存,fst(selected)); }
          缓存内存 = 放入(缓存内存, n,fromJust(s)); }
        否则此处!fetch(n);} }
      _ =>{ 此处.广播(n);
          此!fetchBl(n);} } }

单元 广播(地址 n){
  等待 总线!锁();
  总线.发送读请求(此处, n);
  总线.release(); }
```

图 29. 带注释的获取方法。

```
Maybe<状态> swap(地址 n_out, 对<地址,状态> n_in) {
  Maybe<状态> tmp =无;
  分支选择 (查找(缓存内存,n_out)) {
    无 => 跳过;
    Just(无效状态) => 跳过;
    _ => {
      tmp = 查找(缓存内存,n_out);
      缓存内存 =移除键(缓存内存,n_out);
      如果 (fst(n_n无效状态!=_out) {
        缓存内存 =放入(缓存内存, fst(n_无效状态), 无效状态)); } }
  返回 tmp;}
```

图 30. 交换方法。

```
单元 receiveRd(地址 n,IBarrier 开始,IBarrier end) {
  // 查找(cacheMemory,n))==Just(修改状态): Flush-One-Line;
  // 查找(缓存内存,n))!=Just(修改状态): 忽略单行刷新
  等待 开始!同步();
  分支选择 (查找(缓存内存,n)) {
    Just(修改状态) => 此处!刷新(n);
    _ => 跳过;
  }
  结束!同步(); }
```

图31。带注释的 receiveRd 方法。

```
单元 fetchBl(地址 n){
  // 下一级==无 &
  // fst(select(缓存内存, n)) !=n &
  // select(缓存内存, n)==对(          _,修改状态): 获取Bl3;
  分支选择(下一级) {
    Just(下一级缓存) => {
      Maybe<状态> 状态 = 下一级缓存.getStatus(n);
      // 状态!  =无: LC-Fetch-Unblock
      如果 (状态 == 无){
        此!fetchBl(n); }
      否则 {
        这!fetch(n); }
      }
     _ => {
      对<地址，状态>已选择  =select(cacheMemory, n);
      如果 (fst(selected)==n){

       状态 status  =mainMemory.getStatus(n); // 获取块1
        缓存内存  = 放入(cacheMemory,n,status); }
      否则 {
        分支选择 (selected){
          对(selected   _n,修改状态)  =>{
            此!刷新(selected   _n);
            此!fetchW(n,已选择  _n); }
          对(selected   _n, _)  =>{
            状态 status  = 主存.getStatus(n); // FetchBL2
            缓存内存  =移除键(cacheMemory,selected        _n);
            缓存内存  = 放入(cacheMemory,n,status); }
        } } } } }
```

图 32. 带注释的 fetchBl 方法。

```
  单元 fetchW(地址 n，地址 n     _){
   等待 (查找默认值(缓存内存，n          _，无效状态)!=修改状态); //
   此!fetchBl(n);
  }
```

图 33. </style}带注释的 fetchW 方法。

```
单元 刷新(地址 n) {
// 查找(缓存内存,n)!=修改状态: 刷新2;
  分支选择 (查找(缓存内存,n)) {
    Just(修改状态) => {
      主存.setStatus(n,共享状态); //刷新1
      缓存内存  = 放入(缓存内存,n,共享状态); }
    _ => 跳过;
  } }
```

图 34. 带注释的刷新方法。