

异构计算系统缓存一致性协议及其形式化验证

李仁刚^{1,2)} 闫瑞栋²⁾ 郭振华²⁾ 翟季冬¹⁾ 王 丽²⁾
曹 芳²⁾ 高 开²⁾ 张洪伟²⁾ 杨广文¹⁾

¹⁾(清华大学计算机科学与技术系 北京 100084)

²⁾(浪潮电子信息产业股份有限公司体系结构研究部 北京 100875)

摘 要 缓存一致性协议是保障异构计算系统数据正确性的核心机制,其正确性不仅直接决定系统功能的可靠性,更直接影响系统性能表现。不同计算单元差异化的存储结构与访存模式,导致一致性协议设计与验证面临双重挑战。现有的层次化协议转换与翻译框架利用逻辑复杂的代理缓存或翻译器,构建了多种新型复合一致性协议。然而,这些协议因频繁的消息解析、判断与转化操作,增加了协议设计和验证的复杂性,同时引入额外性能开销,导致其扩展性与兼容性受限。基于此,本文提出一种基于统一状态表示的异构计算系统缓存一致性维护方法,包含协议设计和形式化验证两个关键环节。前者支持异构计算单元间数据一致性访问,减少消息处理数量并降低时延;后者采用符号化定理证明方法,分别针对系统状态机无死锁、单写多读等关键属性进行理论验证。为评估协议的有效性,在gem5仿真器上实现了该协议,并在多种访存路径下测试其事务覆盖率与时延性能。此外,在多个SPLASH-2数据集上开展对比测试。实验结果表明,本文所提出协议可支持异构计算单元间数据访问,其运行时长与经典MESI协议的运行时长差距介于2%~4%之间。

关键词 异构计算系统;一致性协议;形式化验证;协议仿真

中图法分类号 TP393; **DOI号** 10.11897/SP.J.1016.2026.01989

A Cache Coherence Protocol for Heterogeneous Computing Systems and its Formal Verification

LI Ren-Gang^{1,2)} YAN Rui-Dong²⁾ GUO Zhen-Hua²⁾ ZHAI Ji-Dong¹⁾ WANG Li²⁾
CAO Fang²⁾ GAO Kai²⁾ ZHANG Hong-Wei²⁾ YANG Guang-Wen¹⁾

¹⁾(Department of Computer Science and Technology, Tsinghua University, Beijing 100084)

²⁾(Department of Architecture Research Department, IEIT SYSTEMS Co., Ltd., Beijing 100875)

Abstract Cache coherence protocols serve as a critical foundation for ensuring data correctness in heterogeneous computing systems. Their correctness underpins not only functional reliability but also significantly influences overall system performance. The divergent storage architectures and memory access patterns across different computing units pose dual challenges for both the design and verification of protocols. Existing hierarchical protocol translation frameworks often rely on logically complex proxy caches or translators to construct composite coherence protocols. However, these approaches increase the complexity of protocol design and verification due to the

收稿日期:2025-10-15;在线发布日期:2026-04-07。本课题得到国家重点研发计划(2024YFE0210800)资助。李仁刚,博士,中国计算机学会(CCF)杰出会员,正高级工程师,主要研究领域为计算机体系结构设计、人工智能等。E-mail: lirg@ieisystem.com。闫瑞栋(通信作者),博士,高级工程师,中国计算机学会(CCF)会员,主要研究领域为计算机体系结构、分布式系统优化等。E-mail: yanruidong@ieisystem.com。郭振华,博士,正高级工程师,主要研究领域为计算机体系结构设计。翟季冬,博士,教授,博士生导师,主要研究领域为编译优化、高性能计算、人工智能等。王 丽,硕士,高级工程师,主要研究领域为计算机体系结构设计。曹 芳,硕士,高级工程师,主要研究领域为计算机体系结构设计。高 开,硕士,主要研究领域为计算机体系结构设计。张洪伟,博士,主要研究领域为计算机体系结构设计。杨广文,博士,教授,博士生导师,主要研究方向为高性能计算。

frequent need for message analysis, decision-making, and transformation. They also introduce additional performance overhead, resulting in limited scalability and compatibility. To address these issues, we propose a cache coherence maintenance method that employs a unified state representation. It consists of two key components: protocol design and formal verification. The former supports consistent data access across heterogeneous computing units, reducing the number of message exchanges and lowering latency. The latter employs symbolic theorem proving to provide theoretical validation for properties such as deadlock freedom and the Single-Writer-Multiple-Reader (SWMR) invariant within the state machine. For performance evaluation, we implement the protocol in the gem5 simulator and test its transaction coverage and latency under various access paths. Furthermore, comparative tests are conducted using benchmarks from the SPLASH-2. Experimental results show that the proposed protocol facilitates efficient data access across heterogeneous units, while introducing only a minimal 2%~4% runtime overhead compared to the classic MESI protocol.

Keywords heterogeneous computing systems; coherence protocol; formal verification; protocol simulation

1 引 言

近年来,以自然语言处理和计算机视觉为代表的深度学习应用取得了显著进展。数据规模与模型复杂度持续增加,导致算力需求指数级增长^[1]。在此背景下,传统单核处理器已无法满足复杂计算任务的要求,异构计算系统已逐渐成为主流计算架构。异构计算系统^[2]是指在单一计算平台中集成多种具有不同指令集和体系架构的计算单元,并通过协同工作机制实现高效计算。典型异构计算系统通常包括中央处理器(Central Processing Unit, CPU)、图形处理器(Graphics Processing Unit, GPU)等。这些计算单元在处理能力、功耗特性及应用场景方面各具优势。得益于高效协同工作机制,系统计算效率可显著提升。

为缓解异构计算系统中各部件之间因速度差异日益扩大而引发的“存储墙效应”,系统通常采用多级缓存(Cache)、分布式内存架构及写缓冲区等复杂的存储层次结构,以隐藏计算单元与主存间的性能差距^[3]。然而,缓存的引入也带来了数据访问不一致的难题。换言之,系统中同一数据副本可能同时存在于多个缓存中。当某一缓存中的数据被修改时,若其他缓存及主存中的副本未同步更新且仍保持有效状态,则会导致缓存之间以及缓存与主存之间的数据不一致。因此,设计高效低时延的缓存一致性协议是实现缓存管理的关键举措^[4]。目前,学术界与产业界已深入开展缓存

一致性协议的设计与验证研究工作,并取得了阶段性突破^[5-7]。

同构计算系统通过共享内存模型实现统一内存地址共享,并利用诸如 MESI(Modified Exclusive Shared Invalid)^[8]等协议维护处理器核心间的缓存数据一致性。系统通常使用总线监听方式维护一致性^[9]。其工作原理如下:首先,每个处理器核心的缓存控制器会监听总线上的所有内存访问请求。如果某个核心修改了自身缓存中的数据,它会将修改请求发送至总线。随后,其他处理器核心的缓存控制器会检测到该请求,并根据需要更新或无效化各自缓存副本。该机制实现简单且硬件开销低,但随着核心数量的增加,总线带宽成为瓶颈,因此不适合大规模系统。

异构计算系统计算单元间的内存架构与访问机制存在显著差异,且遵循独立的缓存一致性协议。早期系统不支持缓存一致性^[10]。以 CPU-GPU 系统为例, CPU 遵循 MESI 协议,而 GPU 采用 Token Coherence 协议进行内存管理。因此, CPU 无法直接访问 GPU 的内存。当 CPU 需要访问 GPU 内存中的数据时,必须先将其内存中的数据显式拷贝至 CPU 的内存,才能进行访问操作。这种间接访问模式增加了数据访问时延,尤其在高频数据交互场景下,频繁的数据拷贝将大量占用系统带宽资源,进而影响其他数据传输及任务的执行效率。

当前,异构计算系统一致性维护普遍采用分层设计模式^[11]。具体而言,每个计算单元内部会部署本地一致性控制器(Local Coherence Controller,

LCC)及协议转换器,系统层面部署全局一致性控制器(Global Coherence Controller, GCC)实现跨设备协同。一致性维护工作流程如下:①本地请求处理阶段。当LCC接收到一致性请求时,首先尝试在设备内部完成请求解析与响应。若本地资源无法满足请求条件(如缓存行状态冲突或数据缺失),则触发请求转发机制。②全局协调阶段。未完成的本地请求将被路由至GCC,由GCC根据拓扑路由规则将请求分发至目标设备的LCC。③跨设备响应阶段。目标LCC完成请求处理后,生成一致性响应报文并返回至GCC。GCC依据事务标识符将响应转发至原始请求方LCC,完成闭环事务处理。上述关键技术挑战在于GCC接口设计与协议转换器的实现。前者需支持异构一致性协议的指令集映射,通过标准化事务接口实现多协议兼容;后者承担本地与全局协议的状态转换功能,包括将本地协议原语

转换为全局事务格式,对全局响应报文进行语义解析与本地状态更新,维护协议转换状态机,以及确保转换过程原子性等。

综上所述,同构与异构计算系统在内存访问机制、一致性管理策略及应用场景等方面存在显著差异(详见表1)。同构系统基于统一指令集的多核处理器架构,通过硬件自动维护的缓存一致性协议实现全局共享内存模型。由于其低时延访问与开发便捷性,它适用于规则并行任务和应用,但其扩展性受限。相比之下,异构系统利用高速互连协议扩展了计算单元内存空间,虽可为人工智能应用等新型负载提供高效计算能力,但也面临着由复杂协议转换与显式数据迁移引发的额外性能开销和更高的编程复杂度。尽管相关技术已取得阶段性突破,但当前异构计算系统仍需在时延敏感性、计算效率与开发成本之间进行权衡^[12]。

表1 计算系统缓存一致性协议对比

对比维度	同构计算系统协议	异构计算系统协议	本文协议
内存访问机制	硬件自动维护、低时延	显式数据迁移、高时延高带宽	直接访问、降低时延与消息处理量
一致性管理策略	总线监听或集中式目录维护	分层代理多协议翻译转换	分布式目录结构统一状态表示
协议复杂度	低(状态与消息类型有限)	高(频繁消息解析与转换)	中等(通过同一状态减少转换开销)
性能开销	低(适合规则并行任务)	高(转换与数据迁移开销大)	中等(减少冗余消息与状态转换)
可扩展性	差(总线带宽限制)	中等(依赖代理与翻译器)	良好(目录结构支持多设备扩展)
验证方法	模拟测试(准确率低)	模型检测(覆盖率低)	形式化验证(定理证明保障正确性)
适用场景	多核处理器系统	异构加速器计算系统	异构加速器计算系统

异构计算系统缓存一致性协议研究主要面临以下三个技术挑战。

(1)缓存一致性协议设计挑战。异构计算单元拥有与其适配的特异性缓存一致性协议,差异化通信模式加剧了协议设计的复杂度^[13]。经典MESI协议采用读写器发起的失效机制强制执行单写多读(Single Writer Multiple Reader, SWMR)不变性。这种机制适用于CPU,但不适用于GPU。究其原因,GPU通常使用非SWMR的自失效协议,实现更为宽松的一致性模型。因此,这种差异迫使研究者设计复杂的协议转换与翻译策略,将多种异构协议整合为一个庞大的复合协议。虽然学术界和产业界已开发出接口或封装器以简化这一过程引入的转换开销,但手动编写协议仍然面临挑战。

(2)目录控制器设计挑战。随着系统规模的不断扩大,计算单元间的数据交互愈加频繁,导致共享缓存信息增多,采用目录结构的方式维护一致性成为必然选择^[14]。然而,差异化协议导致计算单元内

部目录结构和存储内容不尽相同,进而导致一致性请求、响应处理流程也各不相同。为了能够同步处理和排序异构计算单元的内存访问请求,往往需要添加更多的瞬时状态以保证目录中协议状态高效转换。然而,更多的请求事件和瞬时状态极大地增加了集中式协议目录的设计难度,并且复杂状态之间的转换过程也加剧了系统响应时延。因此,如何设计高效的目录控制器,在保证一致性的同时,降低事务处理时延并减少缓存冲突成为难点。

(3)缓存一致性协议验证挑战。系统规模日益扩大、一致性协议多样、互连网络复杂等诸多因素,使得协议正确性与完备性验证面临挑战。例如,在典型的6×6 Mesh系统中,当MESI与NVLink等多协议共存时,需要验证的全局状态呈现超线性增长趋势,这超出了SPIN^[15]、UPPAAL^[16]等传统模型检测工具的处理能力。因此,急需面向异构计算系统的一致性协议高效验证方案^[17]。

针对上述挑战,本文的创新工作主要包括以下

三个方面:

针对挑战一,本文将经典MESI协议的核心思想引入异构计算环境,构建了面向异构计算系统的缓存一致性维护架构,设计了四种一致性协议状态维护组件,并定义了异构计算单元间缓存一致性事务的统一状态表示,以支持异构计算单元间数据快速访问并避免数据搬运,同时降低现有协议转换与翻译开销。

针对挑战二,本文设计了涵盖主机端与设备端目录子表的新型分布式目录结构,定义了异构计算单元间缓存一致性Load(读请求)、Store(写请求)及Replacement(替换请求)事务处理机制,制定了一致性维护组件稳定状态与瞬时状态的迁移规则,确保异构计算单元间数据高效一致性访问。上述方案不仅降低了现有监听协议的广播通信开销,同时解决了集中式目录设计复杂的难题。

针对挑战三,本文提出一种基于符号化定理证明与仿真模拟相结合的验证方法,从形式化验证与实验评估两个维度论证了缓存一致性协议的正确性与有效性。符号化定理证明通过建立协议的形式化模型,并借助数学逻辑与状态空间分析,严格证明了协议在所有可能执行路径下均满足无死锁、单写多读及数据不变性等关键属性。在此基础上,本文引入仿真模拟实验,通过构建可配置的协议仿真平台,在不同负载与应用场景下定量测量协议的执行时延、消息处理数量及平均运行时长等动态性能指标,从而评估协议的有效性。

基于上述工作,本文提出了一致性协议与验证方案,以实现异构计算单元间数据一致性访问。

本文结构如下:第2节介绍本文所需的背景知识;第3节概述一致性协议架构设计方案;第4节阐述一致性协议事务动态处理机制;第5节阐述一致性协议关键属性的形式化验证方法;第6节分析仿真测试与性能对比实验结果;第7节介绍相关研究进展;第8节总结全文。

2 背景知识

本节首先介绍缓存一致性的定义,其次阐述一致性维护所涉及的状态、事务、网络通道等相关概念及其实现方式,以便于理解后续内容。

2.1 一致性协议维护相关概念

根据文献[10],缓存一致性定义包括以下两个重要属性:一是单写多读特性,即同一时刻对于同一

存储单元只能存在一个写操作或多个读操作;二是数值不变性(Data Value Invariant, DVI),即在任何操作开始时,某个存储单元的值都必须与其最近一次读或写操作结束时的值保持一致。

为实现异构计算单元间数据一致性,本文提出了多种一致性维护组件(见下一节),其涉及的相关概念主要包括组件状态(稳定状态和瞬时状态)、事务(请求、响应等)以及虚拟网络通道(请求队列、转发队列、响应队列)。上述概念关系密切,并严格遵循其运行规则。

(1)组件状态定义。一致性维护组件的状态类型可分为稳定状态与瞬时状态。本文引入MESI协议的稳定状态,在此基础上定义了面向异构计算系统的瞬时状态并进行统一表示。因此,本协议包括以下四个稳定状态:

① M(Modified):表示当前副本已被修改为最新值,仅在当前私有缓存中存在唯一副本且与主存中副本不一致,故在替换时需写回主存。

② E(Exclusive):表示当前副本是独占的,仅在该缓存中存在唯一副本且与主存副本一致,因此替换该数据时无需写回主存。

③ S(Shared):表示当前副本是共享的且未被修改,该副本存在于多个私有缓存中,并与主存中副本保持一致,因此替换该数据时无需写回主存。

④ I(Invalid):表示当前副本无效,若需访问该数据,则需从主存或其他私有缓存中获取该数据后方可操作。

基于上述统一状态表示,各一致性维护组件相互关联并共同遵守单写多读等性质。

除上述稳定状态以外,一致性维护过程中还会产生一系列中间瞬时状态,其功能与行为逻辑因控制器组件而异。本文使用符号XY_Z对瞬时状态进行编码,表示缓存块正在从稳定状态X转换到稳定状态Y,且该转换在Z类型事件发生前不会完成。例如,IM_D表示一个缓存块当前处于I状态,当消息D抵达该缓存块,它将转换至M状态。由于涉及的瞬时状态众多,在后续组件详细设计章节仅展示部分瞬时状态,并重点解释它们的工作原理。

(2)一致性事务。该事务是指为完成特定目标而执行的、具有逻辑关联性的操作集合。这些操作通常涉及异构计算单元缓存与主存之间的数据交互,并保证原子性与一致性,以确保数据的正确性。不同类型的一致性控制器所接收和处理的事务类型有所不同。

(3)虚拟网络通道。一致性维护过程中,一条消息的接收可能触发另一条消息的发送,且两者均需要资源分配,此时需考虑因循环资源依赖引发的死锁问题。一致性消息类型可分为响应、转发及请求消息三类。本文为上述每类消息分别设独立的逻辑虚拟网络通道,并使用消息队列的方式实现。

2.2 一致性协议的实现方式

根据数据副本一致性维护方式的不同,通常可将其分为总线监听协议和目录协议两类。

(1)总线监听协议:该协议利用全局广播通信模式实现缓存状态同步,主要适用于读密集且写操作不频繁的计算负载场景^[18]。在该协议中,各计算单元的私有缓存需持续监听总线事务,一旦检测到总线上广播的一致性请求,便根据本地缓存行状态触发预定义响应逻辑。总线介质的物理特性天然保证了事务的原子性与顺序性,使得状态转换逻辑可简化为确定性有限状态机,从而降低了协议设计与实现难度。然而,随着系统规模的增大,总线负载显著

加重,且总线在任意时刻仅允许一个计算单元使用,造成访存请求频繁阻塞,访存时延增加,因而难以满足大规模系统的需求。

(2)目录协议:该协议采用全局公共目录来记录各缓存行的状态信息,广泛应用于数据更新频繁的场景^[19]。一致性消息均通过集中式目录结构进行转发,查询副本拥有者列表,实现高效的点对点消息传输,从而显著减少了广播通信开销。该协议因其具有良好的扩展性,故被当前主流计算架构普遍采用。

图1对上述两类协议进行了对比。其中,蓝色节点代表数据拥有者节点(拥有一个数组且该数组内存储字符a和b),绿色节点代表无数据节点(数组为空),橙色节点代表数据共享节点(数组内容与数据拥有者节点相同)。节点0、节点1、节点2、节点3为处理器缓存,节点4为主存。以下分别考虑两种数据传输情况:(1)从主存到缓存的传输;(2)缓存到缓存的传输。

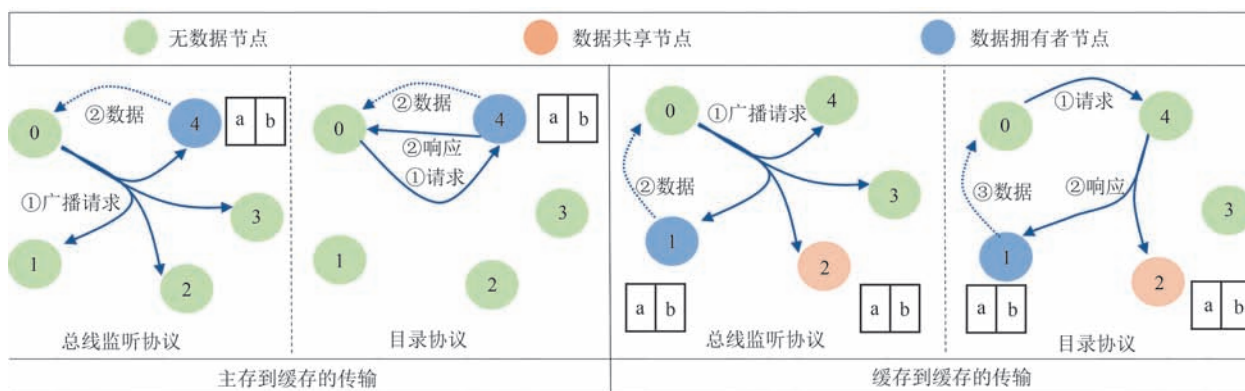


图1 总线监听协议与目录协议对比示意图

对于情况(1),在总线监听协议中,由于请求节点0无法确定数据位于哪个节点,因此首先向所有节点广播数据请求,以获取数据a和b。当数据拥有者节点4收到该请求后,会将完整数据返回给节点0,完成此次请求。而在目录协议中,请求节点0首先向目录所在的节点4发送请求(无需广播)。由于节点4本身即为数据拥有者,它将直接响应请求,并将数据a和b发送至节点0,从而完成数据请求过程。

对于情况(2),在总线监听协议中,节点0仍首先向所有节点广播请求。此时,节点1为数据拥有者,节点2为数据共享者。当节点1或节点2收到广播请求后,会将数据a和b发回请求节点0,完成请求。在目录协议中,请求节点0首先向目录节点4发

送请求。节点4中的目录记录了数据拥有者与共享者的信息,因此将请求转发给数据拥有者节点1和共享者节点2,随后由节点1或节点2将数据发送至请求节点0,最终完成请求处理流程。

3 统一状态表示的缓存一致性协议

本节提出一种基于统一状态表示的缓存一致性协议,旨在实现读写操作的低时延与高效处理。为此,研究将经典的MESI协议引入异构计算环境,构建相应的一致性维护架构及其组件,并定义一致性事务的统一表示方法。通过设计一种新型分布式目录结构,以减轻现有层次化协议在转换与翻译框架中所带来的内存管理开销。此外,协议采用基于目

录结构的方式实现一致性状态的消息传递,在写策略方面选用基于写回机制的写无效模式。

本节将从以下三个层面展开阐述:整体逻辑架构、目录控制器设计以及具体运行实例。具体内容组织如下:3.1节介绍协议的整体逻辑架构及其一致性维护涉及的关键组件;3.2节阐述协议的目录结构设计原理;3.3节与3.4节分别详细说明两类目录控制器的设计细节,涵盖相关状态、事务流程等内容;3.5节通过实例展示各组件在一致性维护过程中的协同工作机制;最后,3.6节对本节内容进行总结。

3.1 一致性协议整体架构

当前同构计算系统中的缓存一致性维护架构通常由分布式局部缓存控制器与全局目录控制器共同构成,二者协同工作以保障数据副本的一致性。然而,为适应异构计算场景中不同类型计算单元在内存访问行为上的差异,需要在原有同构架构的基础上,进一步引入对异构计算单元的支持。为此,本文提出了一种涵盖主机端(如CPU)与设备端(如GPU)的缓存一致性维护架构,如图2所示。相较于传统同构系统,本架构能够支持异构计算单元之间更高效的数据访问,在实现内存扩展能力的同时,显著降低数据搬运带来的开销。

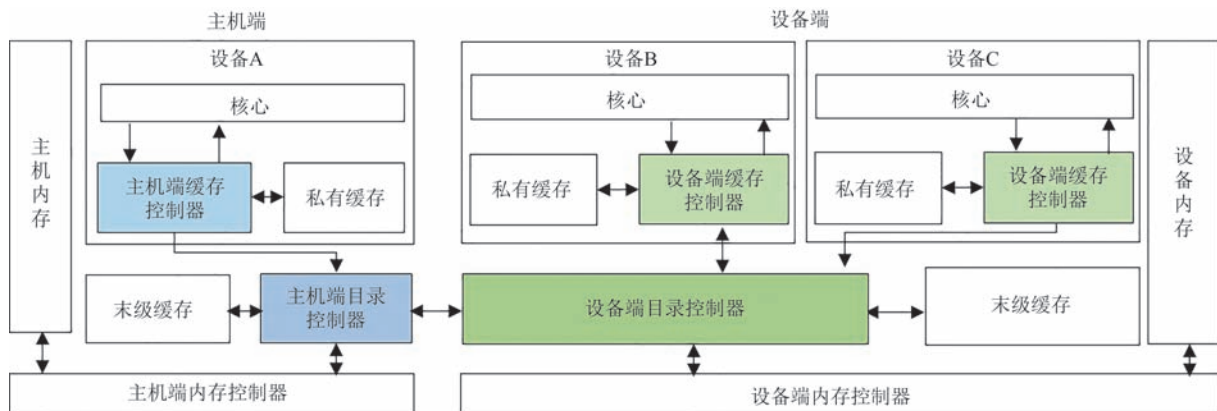


图2 统一状态表示的异构计算系统缓存一致性协议架构

该架构的核心目标是确保异构计算系统中缓存数据的一致性。为此,该架构设计了两类一致性维护组件:缓存控制器与目录控制器。前者细分为主机端缓存控制器(Host Cache Controller, HCC)和设备端缓存控制器(Device Cache Controller, DCC);后者则细分为主机端目录控制器(Host Directory Controller, HDC)和设备端目录控制器(Device Directory Controller, DDC)。下文将简要概述各一致性维护组件的主要功能。

主机端缓存控制器(HCC):每个主机端缓存均配备一个HCC,其核心功能是满足处理器核心对数据的低时延访问需求。当接收到来自处理器核心的读写请求时,HCC首先查询其管理的本地缓存以确认是否命中。在连接关系上,每个HCC与一个处理器核心及其私有缓存直接相连,同时通过系统互连网络与主机端目录控制器通信。

设备端缓存控制器(DCC):每个设备端缓存设有一个DCC,旨在为异构计算单元提供高效的数据访问支持。在收到设备端发起的读写事务时,DCC

优先检查其关联的设备本地缓存。每个DCC连接一个特定的异构计算单元及其对应的设备缓存,并与设备端目录控制器保持连接。

主机端目录控制器(HDC):负责维护所有主机端缓存中数据块的一致性状态,并协调各缓存间的数据同步。在连接层面,HDC不仅通过系统互连网络与所有主机端缓存控制器相连,还连接主机端的未级缓存、内存控制器及设备端目录控制器。

设备端目录控制器(DDC):负责管理设备端缓存中数据块的状态,并维护设备端缓存间的一致性。DDC与设备未级缓存、内存控制器及主机端目录控制器相连,共同构成跨设备一致性维护链路。

3.2 一致性协议目录控制器设计

根据维护数据副本一致性的方式不同,缓存协议通常可分为总线监听和目录协议两大类。总线监听协议依赖于全局广播通信实现缓存状态同步,适用于读密集、写操作较少的计算场景。然而,该协议要求所有私有缓存响应每次访存请求,导致其可扩展

展性受限。相比之下,目录协议通过全局公共目录记录缓存行状态及数据副本的拥有者列表,所有一致性消息经目录结构转发,实现高效的点对点通信,显著降低了广播开销。因此,本文选用目录方式实现异构计算系统的缓存一致性协议。

在基于统一状态表示的缓存一致性维护架构中,HDC和DDC是维护缓存状态同步的核心组件。下文重点阐述这两类目录控制器的结构设计。

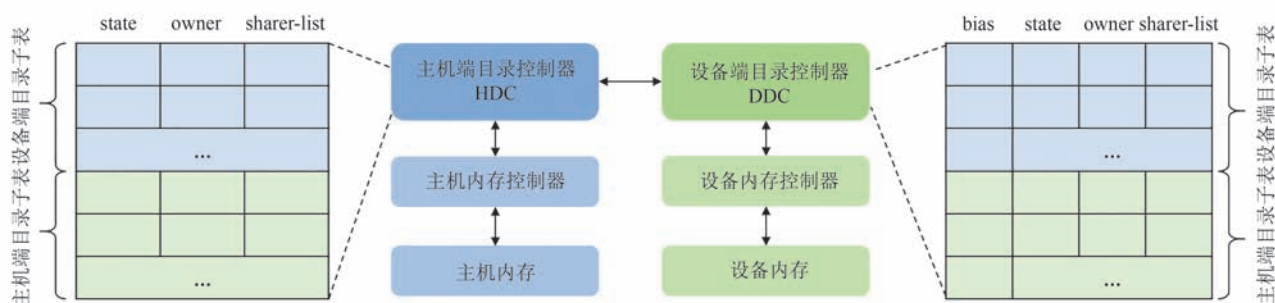


图3 HDC和DDC的目录结构示意图

每个目录子表包含若干条目,每条目对应一个缓存数据块的状态信息,包括以下字段:state、owner、sharer-list及bias。state标记数据块的一致性状态,仅包含M、E、S、I四种稳定状态。owner标识数据块的所有者(仅适用于M或E状态)。sharer-list以one-hot位向量编码并标识数据块的共享者(仅适用于S状态)。bias标识设备偏置信息,占用1个比特位。标记为偏置的设备可直接访问本地内存,无需与主机端交互。

不失一般性,假设主机端有 m 个处理器并且设备端有 n 个设备,则state字段在主机端和设备端均需2比特;owner字段分别需要 $\log_2 m$ 比特(主机端)和 $\log_2 n$ 比特(设备端);sharer-list字段则分别需要 m 比特(主机端)与 n 比特(设备端)。

现给出目录存储空间开销的估算实例。假定主机端处理器(如CPU)与设备端异构计算单元(如GPU)数量均为32,每个处理器与计算单元的末级缓存容量为40 MB,缓存行大小为64 B。依据所设计的目录结构进行计算,主机端目录的存储开销约为95.21 MB,设备端目录的存储开销约为97.66 MB,系统总目录开销合计约为192.87 MB。

目录条目命名规则如下:若某数据块在主机端所有缓存中均处于I状态,则在HDC中其条目状态标记为I;若在主机端一个或多个缓存中处于S状态,则在HDC中标记为S;若在主机端某缓存中处

本协议支持主机端同时缓存来自主机端和设备端内存中的数据,同时也支持设备端缓存来自双方内存的数据。HDC和DDC需同时维护与主机端和设备端缓存相关的信息,其结构如图3所示。两者均由两部分组成:一是主机端目录子表,用于存储来自主机端绑定内存的缓存数据块状态信息;二是设备端目录子表,用于存储来自设备端绑定内存的缓存数据块状态信息。

于M状态,则在HDC中标记为M。设备端目录控制器DDC的条目命名遵循相同原则。

3.3 主机端目录控制器设计

本节介绍主机端目录控制器(HDC)组件所涉及的瞬时状态、事务与虚拟网络通道。

(1)瞬时状态:在数据维护一致性过程中,为精确刻画主机端目录控制器(HDC)组件在不同维护阶段下的动态行为与中间形态,系统定义了一系列瞬时状态。这些状态用于描述该组件在达成最终稳定状态前所经历的短暂且特定的中间情形,主要包括IS_D、IM_D、SS_D、SI_A等共计37个状态。表2展示了部分瞬时状态及其语义说明。

(2)事务:HDC涉及的相关事务是各组件为达成数据一致性而进行的关键交互操作。除瞬时状态外,本小节还定义了56个事务,可归纳为请求、响应与转发三类消息。例如,事务GetS_host表示主机端发起的获取数据共享副本的请求,而事务Data_host则表示主机端缓存控制器的数据响应。如表3所示,这些事务与前述瞬时状态紧密配合,共同构成一个完整且可验证的分布式状态机模型,确保了数据一致性维护过程的严谨性。

(3)虚拟网络通道:为避免一致性协议中潜在的通信死锁风险,本文设计并实现了一种基于虚拟网络(Virtual Networks)通道的消息传输机制。该机制的核心思想是依据消息类型(请求、转发、响应)将其隔离至相互独立的网络通道中进行传输。具体而

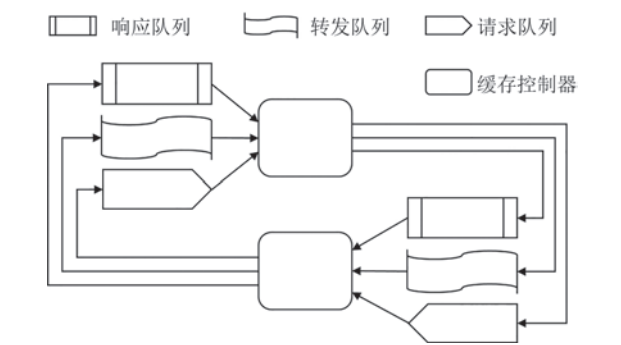
表 2 主机端目录控制器 HDC 涉及的部分瞬时状态举例

序号	瞬时状态	语义描述
1	IS_D	初始状态为 I, 目标状态为 S, 等待数据 Data
2	IM_D	初始状态为 I, 目标状态为 M, 等待数据 Data
3	SS_D	初始状态为 S, 目标状态为 S, 等待数据 Data
4	SI_A	初始状态为 S, 目标状态为 I, 等待无效确认消息 InvAck
5	MS_DW	初始状态为 M, 目标状态为 S, 等待数据 Data 和响应消息 Cmp
6	MI_W	初始状态为 M, 目标状态为 I, 等待响应消息 Cmp
7	...	...

表 3 主机端目录控制器 HDC 涉及的部分事务举例

序号	事务	类型	语义描述
1	GetS_host	请求	来自 HCC 的读数据请求, 目的是请求获取处于 S/E 状态的数据块
2	GetMFromIS_host	请求	来自 HCC 的写数据请求, 请求者处于 S 状态, 且主机端只有一个处于 S 状态的缓存控制器, 目的是请求获取处于 M 状态的数据块
3	Data_host	响应	来自主机端缓存控制器(owner)的数据响应
4	Cmp_dev	响应	来自 DDC 的写回确认响应, 表明之前的数据写回请求已被正确处理
5	RdAny_dev	请求	来自 DDC 的读请求, 目的是获取处于 M/E/S 状态的数据块
6	RdOwn_dev	请求	来自 DDC 的读请求, 目的是获取处于 M/E 状态的数据块
7	...	...	...

言,系统为每类消息分别设立了专属的请求网络、转发网络及响应网络通道,并规定了严格的优先级策略:响应通道优先级最高,其次为转发通道,请求通道优先级最低。如图 4 所示,此种设计确保了高优先级的响应消息不会被低优先级的请求或转发消息阻塞,能够无阻碍地抵达目的节点并被处理,从而切断了可能导致死锁的循环等待链。因此,这种基于消息类别与优先级划分的虚拟网络通道架构,从通信链路层面有效规避了死锁的产生,增强了协议的整体鲁棒性。



基于上述虚拟网络通道的设计思想,HDC 构建了包含三个输入通道与四个输出通道的通信接口,用于实现与主机端一致性控制器及设备端目录控制器的高效、无死锁交互。其三个输入通道分别负责

接收来自主机端内存控制器的内存响应消息,以及来自 HCC 或 DDC 的响应消息与请求消息;四个输出通道则用于向 HCC 或 DDC 发送响应、转发及请求消息,并向主机内存控制器下发内存访问请求。值得注意的是,系统中所有控制器(如 HCC、DCC、DDC)均可依据其交互对象灵活配置相应的消息队列。这些分布在各控制器的输入、输出通道通过特定规则相互连接,共同构建起一个层次清晰、逻辑严密的系统级一致性通信网络,为协议的正确执行提供了底层基础保障。

3.4 设备端目录控制器设计

本节介绍设备端目录控制器(DDC)组件涉及的瞬时状态、事务与虚拟网络通道。

(1)瞬时状态:在设备端目录控制器(DDC)的一致性维护中,瞬时状态被精确定义以刻画缓存块副本在状态转换过程中的瞬时、非稳定情形。这 39 个状态(如 II_m、SS_m、SI_Sm、SI_m 等)共同构成了描述协议行为的形式化基础,确保了状态机模型的严谨性。例如,II_m 表示副本正处于无效状态且已发起向独占状态的迁移请求,而 SS_m 则代表其处于共享状态且正接收其他节点的共享请求。如表 4 所示,对这些瞬时状态的清晰界定与语义说明,是实现协议正确性验证与并发冲突可追溯分析的关键前提,对提升分布式缓存系统的一致性保障能力具有重要意义。

表4 设备端目录控制器DDC涉及的部分瞬时状态举例

序号	瞬时状态	语义描述
1	II_m	初始状态为I,目标状态为I,等待数据响应消息 MemData或响应消息 MemAck
2	SS_m	初始状态为S,目标状态为S,等待数据响应消息 MemData
3	SI_Sm	初始状态为S,目标状态为I,等待监听响应消息 RspIHitSE 和数据响应消息 MemData
4	SI_S	初始状态为S,目标状态为I,等待监听响应消息 RspIHitSE
5	ES_m	初始状态为E,目标状态为S,等待数据响应消息 MemData
6	MS_m	初始状态为M,目标状态为S,等待响应消息 MemAck
7	...	...

(2)事务:在设备端目录控制器(DDC)的一致性协议中,事务是响应内部事件或外部消息而触发、并驱动缓存状态迁移的原子操作单元。本文共定义了37个事务并将其归纳为响应、请求及转发三个类别。例如 GetS_dev代表本地设备发起获取数据共享副本的请求事务,MemData_dev则表示来自设备

内存控制器的数据响应。这些事务与前述瞬时状态相互关联,共同构成一个形式化的状态机模型。如表5所列,对每项事务的精确界定与语义说明,为协议的逻辑正确性验证及运行时行为分析提供了关键依据,确保了异构计算系统在并发访问下的缓存数据一致性。

表5 设备端目录控制器DDC涉及的部分事务举例

序号	事务	类型	语义描述
1	GetS_dev	请求	来自DCC的读数据请求,目的是请求获取处于S/E状态的数据块
2	PutEOwner_dev	请求	来自DCC(为owner)的PutE请求,目的是将缓存中处于E状态的数据块置为无效I,通知DDC清除owner标识
3	MemData_dev	响应	来自设备内存控制器的数据响应
4	DataFromHDC_host	响应	来自HDC的数据响应
5	Data_host	响应	来自DCC的数据响应
6	MemWrSnp_dev	请求	来自HDC的写数据请求,目的是将数据写回设备端绑定内存
7	GoS_host	响应	来自HDC的GO响应,表明数据块应在DDC中应处于S状态
8	MemRdFwdS_dev	转发	来自HDC的转发消息,表明当前数据块在主机端状态为S,数据块可以直接从设备绑定内存转发到设备缓存,而无需向主机进行任何通知
9	...	...	...

(3)虚拟网络通道:设备端目录控制器(DDC)通过精心设计的虚拟网络通道实现与DCC、HDC及设备端内存控制器的解耦与高效协同。该通信架构包含四个输入通道与三个输出通道,构成了DDC处理一致性事务的通信基础。输入通道分别负责接收来自设备端内存控制器的内存响应、来自HDC/DCC的响应与请求以及来自HDC的转发消息,实现了消息的分类异步接收。输出通道则用于向HDC发送响应、向控制器HDC/DCC发起请求以及向设备端内存控制器下发内存操作命令。这种基于消息队列的通道化设计,有效隔离了不同优先级和类型的通信流,减少死锁的发生,为保障一致性维护过程的有序性和低时延提供了关键支撑。

3.5 案例学习

图2展示了异构计算系统中采用统一状态表示的缓存一致性协议框架,旨在从逻辑层面呈现协议

整体架构,并明确数据一致性维护过程中所涉及的关键组件。在该框架中,主机端目录控制器(HDC)与设备端目录控制器(DDC)由于它们所处理的事务不同,因此在设计与实现上存在显著差异。

一方面,HDC负责维护主机端所有缓存块的状态信息,并协调各缓存之间的数据一致性;而DDC则承担设备端缓存状态的维护职责,确保设备内部缓存数据的一致性。另一方面,HDC与DDC之间还需协同维护主机与设备之间的全局数据一致性。为实现上述两个层面的一致性保障,本文分别为HDC与DDC设计了定制化的事务机制与瞬时状态集合。这意味着,两者所涉及的具体事务类型与瞬时状态在结构与数量上均不相同。详细的状态设计可参见附录1~4,其中HDC包括37个瞬时状态、56项事务,以及DDC包括39个瞬时状态、37项事务。

为深入理解HDC与DDC的差异,本小节提供

以下两个实例(实例1和实例2),旨在阐明二者在处理同类请求时所采用的差异化机制。

实例1. 主机端访问设备端内存的访存路径下 Load 读数据请求处理过程。

图5展示了在异构计算系统中,主机端访问设备端内存路径执行 Load 请求时,各组件间的消息传递与状态变迁过程。图中各竖线代表一致性维护组件及其状态时序变化,组件名称标注于竖线上方。组件间的交互消息以带标识的有向箭头表示,标识内容为相应消息的名称。此外,图中亦标注了各一致性组件的状态变化情况。例如“ $I \rightarrow IS_D$ ”表示该组件由状态I迁移至瞬时状态 IS_D 。

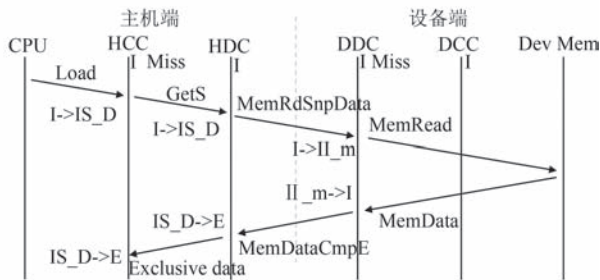


图5 实例1中各一致性维护组件的事务处理时序图

图5各组件初始状态均为I。执行过程始于CPU核心向HCC发出Load请求。由于HCC初始状态为I,发生缓存未命中(即Miss),随即向HDC发送GetS请求,旨在从其他缓存中获取数据副本;同时,HCC状态转为 IS_D 。HDC初始状态同样为I,表明主机端并无该数据的有效缓存副本。HDC在接收到GetS请求后,向DDC发送MemRdSnpData请求,目的是从设备端获取数据,其自身状态亦转为 IS_D 。DDC初始状态为I,表明设备端缓存中亦不存在该数据的有效副本(即未命中)。收到MemRdSnpData请求后,DDC向设备内存发送MemRead请求以读取数据,其状态随之转为 II_m 。设备内存返回携带数据的MemData响应至DDC;DDC继而将附带数据的MemDataCmpE(提示HDC状态变为E/M)响应传回HDC,其状态在此过程中恢复为I。HDC收到响应数据后,向HCC返回带有数据的ExclusiveData消息(HDC的独占数据响应,提示HCC状态即将变为E/M),其状态转为E,表明此时主机端有且仅有一个缓存持有该数据的有效副本。最终,HCC将接收到的数据提供给CPU核心,其状态由 IS_D 转为E,完成本次Load请求的全过程。

实例2. 设备端访问主机端内存的访存路径下 Load 读数据请求处理过程。

图6展示了设备端访问主机端内存路径执行 Load 请求时,各组件之间的消息传递与状态变迁过程。具体而言,设备首先向DCC发出Load请求。DCC初始状态为I,发生缓存未命中(Miss),随即向DDC发送GetS请求,旨在从其他缓存中获取数据副本,同时其自身状态转变为 IS_D 。DDC初始状态亦为I,表明设备端无此数据的有效缓存副本;在收到GetS请求后,DDC进一步向HDC发送RdAny请求(目的是获取处于M/E/S状态的数据块),以从主机端获取数据,并将状态转为 IS_GD 。HDC初始状态为I,表明主机端亦无该数据的有效缓存。收到RdAny请求后,HDC向主机内存发送内存读MemRead请求,以读取对应数据,同时其状态转为 II_m 。主机内存随后返回携带数据的MemData响应至HDC;HDC继而向DDC回复GO-E(提示数据块在DDC中状态置于E)及附带数据的ExclusiveData响应消息(独占数据),其状态由 II_m 恢复为I。DDC在收到响应后,向DCC转发带数据的ExclusiveData消息,并将状态转为E表示此时设备端有且仅有一个缓存持有该数据的有效副本。最终,DCC将数据返回至设备,其状态也转为E,完成本次请求处理流程。

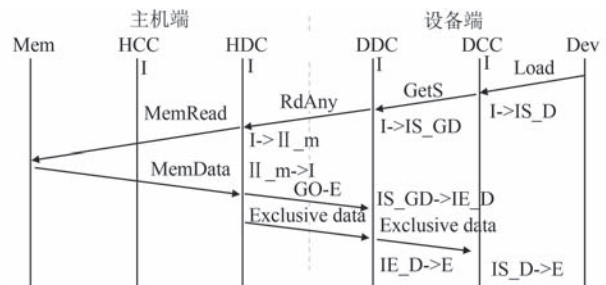


图6 实例2中各一致性维护组件的事务处理时序图

综上所述,HDC与DDC在处理同类请求时,各自状态转换路径与事务逻辑存在明显差异。

3.6 小结

本章系统地阐述了异构计算环境下缓存一致性协议的整体架构及其核心组件的工作原理。在整体架构层面,详细介绍了以主机端缓存控制器(HCC)、主机端目录控制器(HDC)、设备端目录控制器(DDC)及设备端缓存控制器(DCC)为核心的四个组件,并对其各自功能定位及在一致性维护过程中所涉及的关键要素(如瞬时状态、事务类型等)进行了定

义。在详细设计层面,重点剖析了HDC与DDC的目录结构、瞬时状态迁移机制及具体案例。此外,为从根本上规避协议死锁风险,本章提出了一种基于消息类型的系统级虚拟网络通道机制,通过为请求、转发及响应消息建立独立的传输路径与优先级策略,确保了系统在并发环境下能够可靠、无阻塞地运行。

4 一致性协议事务动态运行处理机制

本文的缓存一致性协议可完整支持异构计算环境下的四种典型访存路径,具体包括:主机端访问设备端绑定内存、设备端访问设备端绑定内存、主机端访问主机端绑定内存,以及设备端访问主机端绑定内存。为清晰阐述协议的事务处理机制,本节以“主机端访问设备端绑定内存”这一路径为例,系统说明其请求发起、消息传递与一致性维护的全过程,重点

剖析HCC、HDC、DCC及DDC等控制器在事务处理中的状态迁移逻辑与协同行为。

在该路径中,主机端处理器核心发出的请求主要面向统一状态表示的异构计算系统,包括Load读操作、Store写操作及Replacement替换操作。下文将分类详述各类请求的处理流程。

4.1 统一状态表示的Load读请求处理

需特别说明的是,若主机端缓存控制器(HCC)的初始状态为M、E或S,则视为缓存命中。换言之,HCC中存在所需数据,可将该数据直接返回给主机端下发请求的核心,而无需其他一致性组件参与。因此,这类情况不予讨论。本小节重点讨论HCC初始状态为I时的一致性协议事务处理过程,以揭示协议在缓存未命中场景下的关键行为。

具体而言,主机端处理器核心下发的Load读请求可细分为以下四种情况,如表6所示。

表6 主机端处理器核心下发Load请求后各组件状态汇总分析表

序号	HCC 状态	HDC 状态	DDC/DCC 状态	解释说明
<1>	I	I	I	HCC、HDC、DDC均未命中
			S	HCC、HDC未命中,DDC命中且共享
			E	HCC、HDC未命中,DDC命中且独占
			M	HCC、HDC未命中,DDC命中且修改
<2>	I	S	I	HCC未命中,HDC命中且共享,DDC未命中
			S	HCC未命中,HDC命中且共享,DDC命中且共享
<3>	I	E	I	HCC未命中,HDC命中且独占,DDC未命中
<4>	I	M	I	HCC未命中,HDC命中且修改,DDC未命中

- 情况<1>:HCC 状态为I,HDC 状态为I,则DDC 状态可能为M/E/S/I。
- 情况<2>:HCC 状态为I,HDC 状态为S,则DDC 状态可能为S或I。
- 情况<3>:HCC 状态为I,HDC 状态为E,则DDC 状态只能为I。

- 情况<4>:HCC 状态为I,HDC 状态为M,则DDC 状态只能为I。

针对情况<1>的实例分析:在HCC与HDC初始状态为I并且DDC与DCC状态均为M的条件下,Load请求的完整执行流程以及组件状态迁移情况如图7所示。

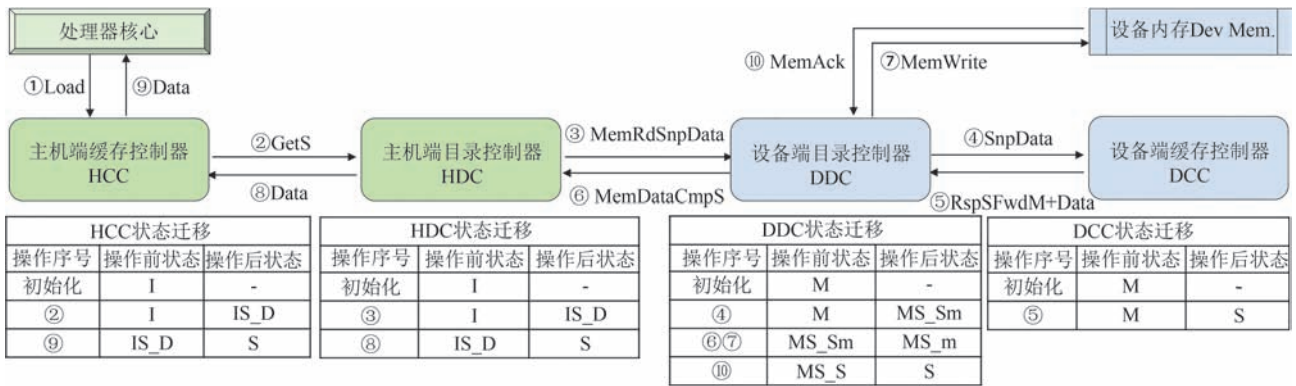


图7 Load请求条件下控制器状态迁移图

算法 1. Load 请求下一致性组件状态迁移情况
输入:处理器核心向缓存控制器HCC发起Load请求。
输出:Load请求处理流程与一致性组件状态变化情况。
初始化:HCC与HDC初始状态为I,DDC与DCC初始状态为M。

- 步骤 1. 若HCC对应目标数据块初始为无效状态I,则触发缓存未命中事件;
- 步骤 2. HCC向HDC发送读数据的GetS请求,其状态由稳态I迁移至瞬态IS_D;
- 步骤 3. HDC在I状态下接收该请求后,通过读数据MemRdSnpData指令(目的是从设备端获取数据,并将设备端处于M/E的状态置为S)向设备端DDC发起数据获取操作,同时HDC状态转换为瞬态IS_D;
- 步骤 4. DDC在独占修改M状态下处理MemRdSnpData请求,执行读操作获取最新数据,具体过程如下:
- 4.1 DDC向DCC发送监听数据SnpData指令,目的是从owner处获取共享数据,DDC状态由M转换为瞬态MS_{Sm};
- 4.2 DCC在M状态响应监听请求,通过携带数据的RspSFwdM响应消息(表明设备缓存在M状态时被命中,即将转换为S状态)将数据块传输至DDC,其状态变更为共享S状态;

- 步骤 5. DDC接收数据后执行以下两个操作:
- 5.1 通过MemDataCmpS响应(提示HDC的状态即将转换为S)将数据回传至HDC;
- 5.2 同时向设备内存控制器发起MemWrite写回请求,而后DDC状态由MS_{Sm}更新为MS_m;
- 步骤 6. HDC完成MemDataCmpS数据接收后,将数据转发至初始请求方HCC,其状态由IS_D升级为S稳态;
- 步骤 7. HCC获取数据后最终完成状态由IS_D至S的转换,实现数据可用。注意到当DDC接收到内存控制器MemAck确认后,其状态将由MS_m最终稳定为S状态。

算法 1 描述了图 7 中的各一致性维护组件在 Load 请求条件下的事务与状态迁移情况。HCC 组件由最初的稳定状态 I 最终转换为状态 S,并将读取的数据返回至下发请求的处理器核心,完成整个读请求执行过程。

上述流程通过多个一致性维护组件协同工作,实现了缓存一致性协议中数据获取、状态迁移和内存同步等操作。

4.2 统一状态表示的 Store 写请求处理
主机端处理器核心下发的 Store 写请求可细分为以下五种情况,如表 7 所示。

表 7 主机端处理器核心下发 Store 请求后各组件状态汇总分析表

序号	HCC 状态	HDC 状态	DDC/DCC 状态	解释说明
1	I	I	I	HCC、HDC、DDC 均未命中
			S	HCC、HDC 未命中,DDC 命中且共享
			E	HCC、HDC 未命中,DDC 命中且独占
			M	HCC、HDC 未命中,DDC 命中且修改
2	I	S	I	HCC 未命中,HDC 命中且为共享,DDC 未命中
			S	HCC 未命中,HDC 命中且共享,DDC 命中且共享
3	S	S	I	HCC、HDC 命中且共享,DDC 未命中
			S	HCC、HDC 命中且为共享,DDC 命中且共享
4	I	E	I	HCC 未命中,HDC 命中且独占,DDC 未命中
5	I	M	I	HCC 未命中,HDC 命中且修改,DDC 未命中

- 情况 1:HCC 状态为 I,HDC 状态为 I,则 DDC 状态可能为 M/E/S/I。
- 情况 2:HCC 状态为 I,HDC 状态为 S,则 DDC 状态可能为 S 或 I。
- 情况 3:HCC 状态为 S,HDC 状态为 S,则 DDC 状态可能为 S 或 I。
- 情况 4:HCC 状态为 I,HDC 状态为 E,则 DDC 状态只能为 I。
- 情况 5:HCC 状态为 I,HDC 状态为 M,则 DDC 状态只能为 I。
- 针对情况 1 的实例分析:在 HCC、HDC、DDC 以及 DCC 状态均为 I 的条件下,Store 请求的完整事

- 务流程及其状态转换机制如图 8 所示。
- 算法 2.** Store 请求下一致性组件状态迁移情况
输入:处理器核心向缓存控制器HCC发起Store请求。
输出:Store请求处理流程与一致性组件状态变化情况。
初始化:HCC、HDC、DDC及DCC初始状态为I。
- 步骤 1. 若HCC对应目标数据块初始为无效状态I,则触发缓存未命中事件;
- 步骤 2. HCC向HDC发送独占修改请求GetM,其内部状态由稳态I转换为中间瞬态IM_D;
- 步骤 3. 当HDC在I状态接收该请求后,通过读监听无效MemRdSnpData指令(目的是从设备端获取数据,并将设备端处于M/E/S的状态置为无效)向DDC发起数据副本获

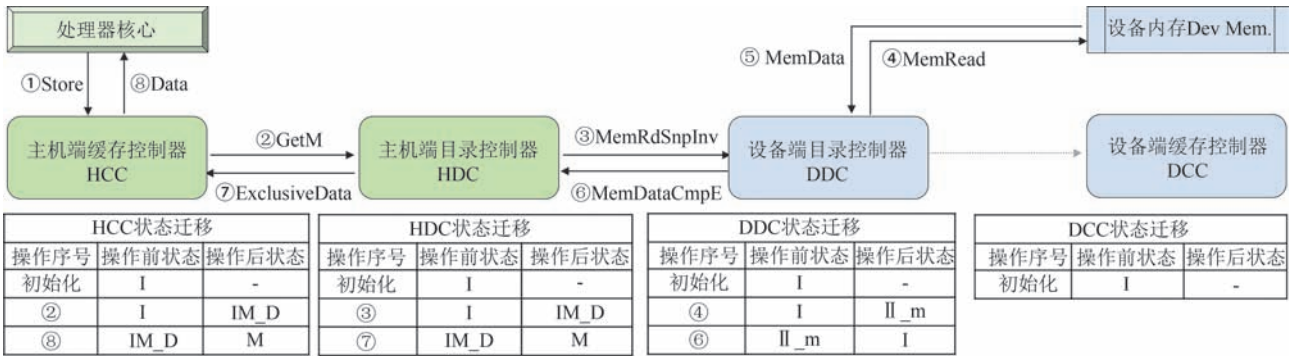


图8 Store 请求条件下控制器状态迁移图

取并执行缓存行失效操作,同时HDC 状态迁移至IM_D 中间状态;

步骤4. DDC 在I 状态处理MemRdSnpiInv 请求时,因设备缓存中无有效数据副本,直接向设备绑定内存发起内存读MemRead 请求,状态由I 转换为II_m 中间态;

步骤5. 设备内存控制器响应MemRead 请求后,通过MemData 响应将数据回传至DDC;

步骤6. 数据接收完成后,DDC 利用MemDataCmpE 响应将数据包转发至HDC(提示HDC 的状态即将变为E/M),状态由II_m 恢复为初始I 状态;

步骤7. HDC 解析MemDataCmpE 响应后确认设备缓存无有效数据,将独占数据通过ExclusiveData 消息(HDC 的独占数据响应,提示HCC 状态即将变为E/M)传输至初始

请求方HCC,其状态由IM_D 升级为M 稳态。

步骤8. 最终事务阶段,HCC 接收ExclusiveData 数据后完成状态由IM_D 至M 的转换,HCC 建立缓存独占所有权。

上述流程通过多个一致性维护组件协同工作,实现了写操作中缓存一致性协议要求的独占状态获取、数据有效性维护及内存数据同步等核心功能。本协议通过MemRdSnpiInv 指令的读监听无效机制,在获取内存数据的同时确保设备缓存副本的状态一致性,展现了异构缓存管理的技术优势。

4.3 统一状态表示的Replacement 替换请求处理

主机端处理器核心触发Replacement 请求可分为以下四种情况,如表8 所示。

表8 主机端处理器核心触发 Replacement 替换请求后各组件状态汇总分析表

序号	HCC 状态	HDC 状态	DDC/DCC 状态	解释说明
1	S(多个)	S	I	HCC、HDC 命中且共享,DDC 未命中
			S	HCC、HDC、DDC 命中且共享
2	S(唯一)	S	I	HCC、HDC 命中且共享,DDC 未命中
			S	HCC、HDC、DDC 命中且共享
3	E	E	I	HCC、HDC 命中且独占,DDC 未命中
4	M	M	I	HCC、HDC 命中且修改,DDC 未命中

情况1:HCC 状态为S(多个),HDC 状态为S,则DDC 状态为S或I。

情况2:HCC 状态为S(唯一),HDC 状态为S,则DDC 状态为S或I。

情况3:HCC 状态为E,HDC 状态为E,则DDC 状态只能为I。

情况4:HCC 状态为M,HDC 状态为M,则DDC 状态只能为I。

以情况4 为例,本部分进行分析并阐述各控制器的状态迁移过程,如图9 所示。

算法3. Replacement 请求下一致性组件状态迁移情况

输入:处理器向控制器HCC1 发起Replacement 请求。

输出:Replacement 请求中一致性组件状态变化情况。

初始化:HCC1 与HDC 初始状态为M,HCC2、DDC 及DCC 初始状态为I。

步骤1. 若HCC1 对应目标数据块初始为独占修改状态M,则HCC1 将向HDC 发送PutM 写回内存请求,其内部状态由稳态M 迁移至中间态MI_A;

步骤2. HDC 在M 状态接收请求后,基于数据块的修改属性判定需执行内存回写操作,遂向DDC 发送携带数据的MemWrSnpi 请求(HDC 的写数据请求,目的是将数据写回设备端内存),同时向HCC1 返回PutAck 确认响应,其状态由M 转换为中间态MI_W;

步骤3. HCC1 接收PutAck 响应后立即执行缓存行失效操作,状态由MI_A 降级为无效I ;

步骤4. DDC 在I 状态处理MemWrSnpi 请求时,通过

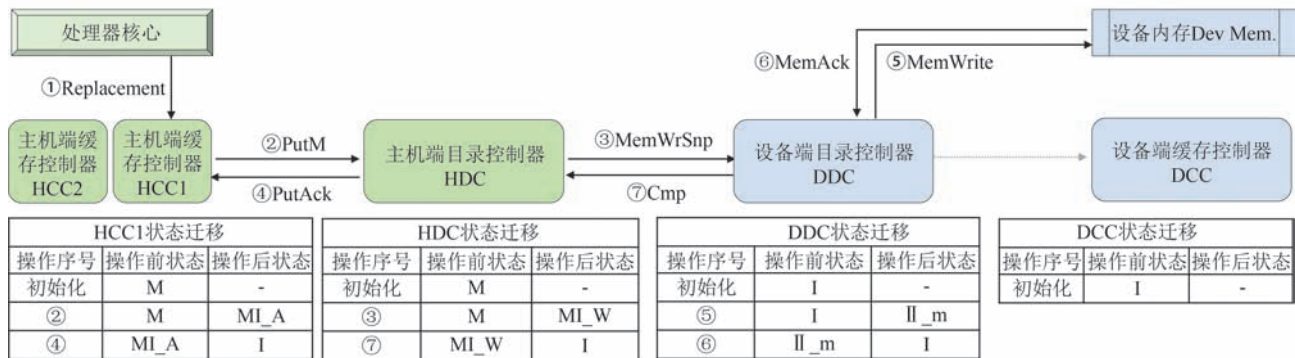


图9 Replacement请求条件下控制器状态迁移图

MemWrite 指令将数据块写入设备绑定内存,状态由 I 转换为中间态 II_m;

步骤5. 内存控制器完成数据操作后,通过 MemAck 响应触发 DDC 状态由 II_m 恢复至 I,并向 HDC 发送 Cmp 信号(DDC 的写回确认响应,表明之前发出的数据写回请求已被正确处理);

步骤6. 最终,HDC 接收 Cmp 信号后完成由 MI_W 至 I 的状态转换,实现数据一致性闭环。

本策略有效管理了修改数据回写过程:HCC 负责协调请求与数据写回指令的并发执行,DDC 专注

内存同步操作,在保障数据可靠性的同时,最大程度优化了缓存替换操作的系统吞吐效率。

4.4 一致性协议事务冲突避免策略

为应对缓存一致性协议事务的冲突,本小节提出基于阻塞队列与非阻塞式两类策略。

(1) 采用阻塞队列的方式解决冲突访问。图 10 分别以蓝色与黑色箭头展示了在主机端与设备端同时发起 Store 写请求时,各组件间的消息传递与状态变迁过程。当上述两类请求并发产生时,将引发访问冲突。

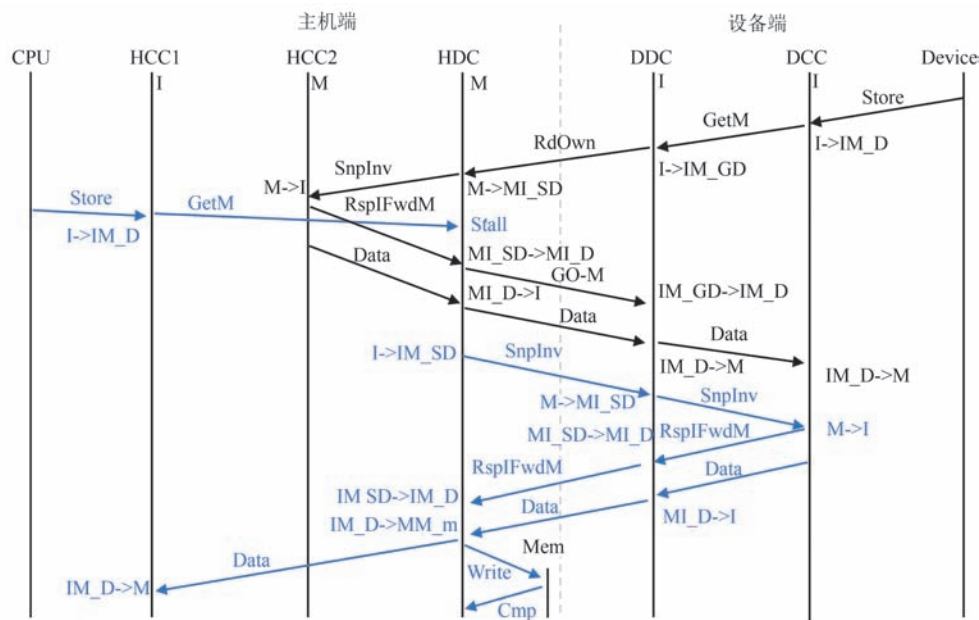


图10 采用阻塞队列方式解决冲突访问的示例

为此,本文提出采用阻塞队列机制以协调此类冲突。在该场景中,设备端首先发出 Store 请求,随后主机端 CPU 核心亦发出 Store 请求。设备端请求经 DCC 与 DDC 处理后,率先抵达 HDC。此时 HDC 初始状态为 M,表明主机端存在一个缓存(如

HCC2) 持有该数据的有效副本。HDC 随即向 HCC2 发送监听无效 SnpInv 消息,同时其自身状态转为瞬时状态 MI_{SD}。此后,主机端 CPU 核心发出的 Store 请求在 HCC1 处未命中,进而向 HDC 发送 GetM 请求。此时 HDC 已处于由设备端请求触

发的 MI_SD 状态。为避免进一步冲突, GetM 请求在此被放入一个阻塞队列中且令其处于等待 (Stall) 状态, 而设备端发起的请求流程则继续执行。待设备端流程执行完毕, HDC 状态转变为稳定状态 I 时, 之前被阻塞的 GetM 请求得以解除阻塞, 并继续沿既定路径 (蓝色箭头) 执行。此后, 主机端与设备端两方的消息处理流程彼此独立执行, 直至完成。因此, 本文所采用的阻塞队列机制能够有效避免并发 Store 请求导致的冲突问题。

(2) 采用非阻塞式的方式解决冲突。图 11 分别描绘了主机端 CPU1 与 CPU2 中两个核心同时发起 Replacement 请求 (黑色箭头) 与 Load 请求 (蓝色箭头) 时, 各组件间的消息传递与状态转换过程。

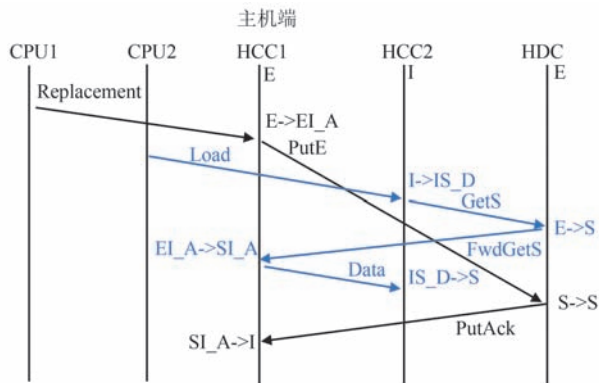


图 11 采用非阻塞方式解决冲突访问的示例

具体而言, CPU1 首先触发 Replacement 请求, 该请求首先抵达 HCC1, 促使其向 HDC 发送 PutE 消息; 与此同时, HCC1 状态由 E 转为瞬时状态 EI_A。在此期间, CPU2 对同一地址发起了 Load 请求, 该请求经 HCC2 与 HDC 处理后, 生成的 FwdGetS 消息 (来自 HDC 的转发请求, 目的是从 owner 处获取共享数据) 亦被转发至 HCC1。此时, HCC1 已因 CPU1 的请求流程进入 EI_A 状态。为提升系统性能, FwdGetS 消息在此并未被阻塞。分析表明, FwdGetS 旨在获取有效数据, 而 HCC1 当前虽处于 EI_A 状态, 其数据副本尚未失效, 仍为有效状态。因此, HCC1 在接收到 FwdGetS 后, 可直接将数据发送至 HCC2, 并将其自身状态由 EI_A 转为 SI_A。随后, 当 CPU1 的请求流程从 HDC 返回 PutAck 确认消息至 HCC1 时, HCC1 基于 SI_A 状态先执行数据失效操作, 继而将状态更新为 I, 从而完成整体协调过程。

综上, 阻塞式方法以性能延迟为代价换取了设计的简洁性, 非阻塞式方法则以更高的实现复杂度

避免了性能损失。鉴于此, 为优化性能, 本文优先选用后者。

4.5 小结

本节以“主机端访问设备端绑定内存时访存路径”为例, 详细阐述协议读、写以及替换请求响应流程设计方案, 并着重解释各控制器组件在其一致性维护事务中状态的变化情况。通过实例化分析, 便于读者理解协议设计的核心思想。

5 一致性协议关键属性形式化验证

为应对异构缓存一致性协议验证的挑战, 本节提出一种基于标记迁移系统 (Labelled Transition System, LTS) 的严格形式化框架。该框架通过构建协议的完整自动机模型, 基于此模型使用计算树逻辑公式定义并验证其关键性质, 从而为协议的正确性提供保证。

5.1 一致性协议抽象与符号建模

缓存一致性协议抽象与建模提供了形式化模型定义, 旨在通过数学符号化的描述方式, 为缓存一致性协议的设计与验证提供严谨的逻辑基础。缓存一致性协议的核心行为模式是“事务驱动的状态转移”。系统中的每个组件 (目录控制器、缓存控制器) 在任何时刻都处于有限个明确定义的状态之一。组件之间的交互 (发送/接收消息) 作为离散的事务或动作, 精确地触发组件从当前状态转移到下一个状态。这种“状态-动作-新状态”的二元关系与标记迁移系统的数学模型十分吻合。因此, 使用 LTS 对协议建模, 不仅能准确描述其动态行为, 还能为后续利用形式化验证工具分析其属性提供标准化的输入模型。下面给出协议模型的关键定义。

定义 1 (缓存协议 LTS 模型的形式化定义). 缓存一致性协议可建模为以下的六元组 $\mathcal{M}$:

$$\mathcal{M} = (\mathcal{S}, \Sigma, \rightarrow, s_0, AP, L).$$

其中, ① $\mathcal{S}$ 表示系统所有可能的全局配置集合。一个全局配置包含了所有组件 (目录、缓存、内存、网络通道) 在某一时刻的快照。

② Σ 表示系统中所有可能发生的事务或动作的集合。这些事务是状态迁移的触发器或标签。

③ $\rightarrow \subseteq \mathcal{S} \times \Sigma \times \mathcal{S}$ 是一个三元关系的集合, 定义了系统如何从一个状态转换到另一个状态。例如, 若 $(s, \sigma, s') \in \rightarrow$, 则记作 $s \xrightarrow{\sigma} s'$, 表示在状态 s

($s \in \mathbb{S}$)下发生动作或事务 σ 后,系统可以迁移到状态 s' 。

④ $s_0 \in \mathbb{S}$ 表示系统开始执行的初始全局状态。

⑤ AP 是描述系统状态局部特征的、最基本的命题集合,即原子命题。它是关于单个组件、地址或消息的简单事实。例如,主机端目录控制器中某个缓存块的状态为 M 。

⑥ $L: \mathbb{S} \rightarrow 2^P$ 表示标签函数。该函数为每个状态 s 分配一个集合 $L(s) \subseteq AP$,这个集合包含了在状态 s 下为真的所有原子命题。

定义 2(状态空间 $\mathbb{S}$ 的形式化定义). 系统全局状态 $s \in \mathbb{S}$ 定义为如下的七元组:

$$s = (C_{HDC}, C_{DDC}, C_{HCC}, C_{DCC}, \text{Mem}, \text{Net}, \text{Tran}).$$

其中,① $C_{HDC}: \mathcal{A} \rightarrow D_{HDC}$ 表示主机端目录控制器状态函数。 $\mathcal{A}$ 表示系统物理地址的有限集合, D_{HDC} 则表示HDC的状态集合。它的具体定义可描述为 $D_{HDC} = \text{Stable} \cup \text{Transient}_{HDC} \times \text{Info}_{HDC}$ 。其中,

$\text{Stable} \in \{M, E, S, I\}$ 和 Transient_{HDC} 分别表示四种稳定状态以及与主机端目录控制器相关的瞬时状态集合。 $\text{Info}_{HDC} = \{\text{owner}: O_{host}, \text{sharers}: 2^{O_{host}}\}$ 表示附加结构信息,并且 $O_{host} = \{CPU_1, \dots, CPU_m\}$ 为主机端处理器表示集合。例如,地址 $a \in \mathcal{A}$ 在HDC中被 CPU_1 独占可描述为:

$$C_{HDC}(a) = (E, (\text{owner} = \text{null}, \text{sharers} = \{CPU_1\})).$$

② $C_{DDC}: \mathcal{A} \rightarrow D_{DDC}$ 表示设备端目录控制器状态函数,其结构与 C_{HDC} 类似,但瞬时状态集合与附加的结构信息与设备端相关。

③ $C_{HCC} = [C_{HCC}^1, \dots, C_{HCC}^m]$ 表示HCC的状态向量。 $C_{HCC}^i: \mathcal{A} \rightarrow C_{cache}$ 且 $C_{cache} = \text{Stable} \cup \text{Transient}_{cache}$,表示某地址在缓存中的状态。例如 $C_{HCC}^1(a) = (S)$,表示 CPU_1 中地址 $a \in \mathcal{A}$ 所在缓存状态为 S 。

④ $C_{DCC} = [C_{DCC}^1, \dots, C_{DCC}^n]$ 表示设备端缓存控制器状态向量,其结构与 C_{HCC} 类似。

⑤ $\text{Mem}: \mathcal{A} \rightarrow \mathcal{V}$ 表示内存状态函数, $\mathcal{V}$ 表示数值的集合。例如 $\text{Mem}(a) = v$,表示内存地址 $a \in \mathcal{A}$ 的值为 $v \in \mathcal{V}$ 。

⑥ $\text{Net} = \{Q_{Req}, Q_{Fwd}, Q_{Resp}\}$ 表示网络通道。其中每个消息队列是对应动作或事务的集合并且遵循先进先出原则。请求队列 Q_{Req} 用于存放请求类消息(如GetS、GetM)。转发队列 Q_{Fwd} 用于存放转发消息(如SnpData)。响应队列 Q_{Resp} 用于存放响应消息(如Data、Cmp)。

⑦ Tran 表示活跃事务标识符集合,用于唯一标

识系统中正在进行的、未完成的一致性事务,以匹配请求和响应。

定义 3(动作集合 Σ 的形式化定义). 动作集合 Σ 被划分为互不相交的请求操作、响应操作以及转发操作,即 $\Sigma = \sum_{Req} \cup \sum_{Resp} \cup \sum_{Fwd}$ 。

其中,① $\sum_{Req}$ 为请求动作,由请求方主动发出,旨在获取数据访问权限或执行特定操作。如 $\text{GetS}(a, i)$ 表示主机端处理器 $i \in O_{host}$ 请求以共享权限 S 访问地址 $a \in \mathcal{A}$ 。

② $\sum_{Resp}$ 为响应动作,表示对请求或转发消息的最终答复。例如, $\text{Ack}(a, src)$ 表示组件 src (如HCC、HDC等)的确认消息。

③ $\sum_{Fwd}$ 为转发动作,表示目录或中间组件为处理请求而发起的协调操作。例如, $\text{FwdGet}(a, tgt)$ 表示将获取共享数据的请求转发至当前拥有者 tgt 。

定义 4(迁移关系 $\rightarrow$ 的形式化定义). 迁移关系通过一组推理规则来定义。每条规则描述在何种条件下,系统可以执行某个动作并迁移到新状态。规则采用以下标准形式:

$$\frac{\text{Pre}(s, \sigma)}{\exists s' \in \mathbb{S}; (s, \sigma, s') \in \rightarrow \wedge \text{Post}(s, s', \sigma)}.$$

其中,横线上方部分称为前提,它是一个或多个关于当前状态 s 和待执行动作 σ 的条件合取。只有当所有前提为真时,这条规则才可能被应用。横线下部分称为结论,它断言如果前提成立,那么存在一个后续状态 s' ,使得三元组 (s, σ, s') 属于迁移关系 $\rightarrow$,并且 s' 满足描述规则。其中 $\wedge$ 表示合取(逻辑与), $\exists$ 表示存在量词。

实例 3(主机端请求未命中且HDC状态为 I).

$$\begin{aligned} & C_{HCC}^i(a) = I \wedge C_{HDC}(a).stab = I \\ & \wedge \neg \exists t \in \text{Tran}; \text{conflict}(t, a) \wedge \sigma = \text{GetS}(a, i) \\ & \hline & \exists s': (s, \sigma, s') \in \rightarrow \end{aligned}$$

$$\wedge C'_{HDC}(a) = (IS_D, (\text{owner} = \text{null}, \text{sharers} = \emptyset))$$

$$\wedge Q'_{Req} = Q_{Req}. \text{MemRdSnpData}(a)$$

$$\wedge \text{Tran}' = \text{Tran} \cup \{t_{new}\}.$$

上述前提第一行表达式说明处理器 i 的HCC中对应地址 a 的状态为 I ,并且HDC中地址 a 的目录条目的稳定状态也为 I ;第二行表达式说明不存在与当前请求地址 a 冲突的活跃事务 t ,且声明要执行的动作为 GetS 。

结论中前两行表达式说明HDC中该地址的状

态转换为瞬时状态 IS_D ; 第三行表达式说明在请求消息队列追加一条发给 DDC 的 $MemRdSnpData$ 读数据请求消息; 第四行表达式说明生成一个新的事务标识符并加入活跃事务集合。

5.2 一致性协议关键属性验证

现有公理化证明与模型检测工具在验证异构计算系统缓存一致性协议时存在局限性。这些方法试图一次性建立完整的全局模型并验证所有属性, 这直接导致了状态空间爆炸问题。基于上述 LTS 模型, 本文使用计算树逻辑 (Computation Tree Logic, CTL) 对协议的若干属性进行独立的形式化验证, 从而在保证严谨性的同时有效降低验证复杂度。

为实现上述目标, 本文并非对所有属性面面俱到, 而是聚焦于构成协议正确性最本质与最核心的约束条件。缓存一致性协议的正确性通常涵盖安全与活性两个方面: 安全属性保证“坏的事情永远不会发生”, 而活性属性保证“好的事情最终会发生”。本文选取的四个验证属性中, 无死锁性属于最基本的活性属性, 确保系统不会因循环资源等待而陷入全局停滞, 目录一致性、单写多读和数值不变性则属于核心的安全属性, 它们共同保障了数据访问的正确性。因此, 上述四个属性被选为形式验证的重点。下面给出公理化验证分析过程。

定理 1. 缓存一致性协议不存在死锁, 即

$$\mathcal{M}, s_0 \models AG \left(\bigvee_{\sigma \in \Sigma} \text{enabled}(\sigma, s) \right).$$

其中, $\models$ 表示满足关系。 $\mathcal{M}, s_0 \models \phi$ 表示在模型 $\mathcal{M}$ 中, 从初始状态 s_0 开始, 时态逻辑公式 ϕ 为真。 $AG(\cdot)$ 表示 CTL 的路径量词与时态算子。 $AG\phi$ 意为在所有路径的所有状态上, 公式 ϕ 都成立。 $\vee$ 表示逻辑或符号。原子命题使能性 (即可发生) 的定义为 $\text{enabled}(\sigma, s) \equiv \exists s' \in \mathbb{S}: (s, \sigma, s') \in \rightarrow$, 即在状态 s 下为真当且仅当动作 σ 在 s 下是使能的。

定理 1 断言从初始状态 s_0 开始, 无论系统沿任何可能的执行路径演化, 在每一个可达状态, 都至少有一个协议定义的动作 (请求、转发或响应) 是能够被立即执行的。换句话说, 系统永远不会进入一个所有动作都无法执行的“全局停滞”状态。为证明定理 1, 本文首先引入如下三个引理。

引理 1. 响应动作满足使能性, 即

$$\forall s \in \mathbb{S}: s \wedge Q_{\text{Resp}}(s) \neq \emptyset \Rightarrow \left(\bigvee_{\sigma \in \Sigma_{\text{Resp}}} \text{enabled}(\sigma, s) \right).$$

引理 1 表明针对任何可达状态 s , 只要响应队列

Q_{Resp} 不为空集 $\emptyset$, 则至少存在一个响应类动作 σ 在状态 s 下是使能的。(引理 1 证明见附录 5)

引理 2. 转发动作满足使能性, 即

$$\forall s \in \mathbb{S}: s \wedge Q_{\text{Resp}}(s) = \emptyset \wedge Q_{\text{Fwd}}(s) \neq \emptyset \Rightarrow \left(\bigvee_{\sigma \in \Sigma_{\text{Fwd}}} \text{enabled}(\sigma, s) \right).$$

引理 2 表明针对任何可达状态 s , 如果响应队列 Q_{Resp} 为空集 $\emptyset$ 并且转发队列 Q_{Fwd} 不为空集 $\emptyset$, 则至少存在一个转发类动作 σ 在状态 s 下是使能的。(引理 2 的证明见附录 6)

引理 3. 请求动作满足使能性, 即

$$\forall s \in \mathbb{S}: s \wedge Q_{\text{Resp}}(s) = \emptyset \wedge Q_{\text{Fwd}}(s) = \emptyset \Rightarrow \left(\bigvee_{\sigma \in \Sigma_{\text{Req}}} \text{enabled}(\sigma, s) \right) \vee \text{Idle}(s).$$

引理 3 表明针对任何可达状态 s , 如果响应队列 Q_{Resp} 和转发队列 Q_{Fwd} 均为空集 $\emptyset$, 要么至少存在一个请求类动作 σ 在 s 下是使能的, 要么系统处于一个空闲状态 $\text{Idle}(s)$ 。(引理 3 的证明见附录 7, 定理 1 的证明见附录 8)

定理 2. 目录条目始终反映缓存真实状态, 即

$$\mathcal{M}, s_0 \models AG(\forall a \in \mathcal{A}: \text{DirC}_{\text{host}}(a) \wedge \text{DirC}_{\text{dev}}(a)).$$

其中, $\text{DirC}_{\text{host}}(a) \equiv C_{\text{HDC}}. \text{stab}(a) = \text{Agg}_{\text{host}}(a)$ 并且 $\text{DirC}_{\text{dev}}(a) \equiv C_{\text{DDC}}. \text{stab}(a) = \text{Agg}_{\text{dev}}(a)$ 。

对于地址 $a \in \mathcal{A}$, 由协议设计原理可知其目录一致性状态 $\text{DirC}_{\text{host}}(a)$ 定义为其在 HDC 的稳定状态 $C_{\text{HDC}}. \text{stab}(a)$ 。而该稳定状态是通过主机端聚合函数 $\text{Agg}_{\text{host}}(\cdot)$ 以及设备端聚合函数 $\text{Agg}_{\text{dev}}(\cdot)$ 聚合所有相关缓存行的状态推导并生成的。 $\text{Agg}_{\text{host}}(\cdot)$ 与 $\text{Agg}_{\text{dev}}(\cdot)$ 的定义对称。下面以 $\text{Agg}_{\text{host}}(\cdot)$ 为例, 介绍其具体形式化描述:

$$\text{Agg}_{\text{host}}(\cdot) = \begin{cases} M, & \text{若 } (\exists i \in O_{\text{host}}: C_{\text{HCC}}^i(a) = M) \\ & \vee (C_{\text{DDC}}(a). \text{stab} = M \\ & \wedge C_{\text{HDC}}(a). \text{Info}. \text{owner} = \text{HDC}); \\ S, & \text{若 } (\exists i, j \in O_{\text{host}}, i \neq j: \\ & C_{\text{HCC}}^i(a) = S \wedge C_{\text{HCC}}^j(a) = S) \\ & \vee (C_{\text{DDC}}(a). \text{stab} = S \\ & \wedge C_{\text{DDC}}(a). \text{Info}. \text{owner} \neq \emptyset); \\ E, & \text{若 } \exists! i \in O_{\text{host}}: C_{\text{HCC}}^i(a) = E \\ & \wedge \forall j \in O_{\text{dev}}: C_{\text{DDC}}^j(a) \notin \{S, M\}; \\ I, & \text{否则。} \end{cases}$$

其中符号 $\exists!$ 表示存在唯一。上述定义表明在任何时候,HDC中记录的状态都必须精确地反映其管辖范围内所有缓存中的真实状态聚合结果。例如,如果有一个主机端HCC中缓存行处于M状态,那么HDC中该地址的状态必须为M;如果有多个缓存行处于S状态,则HDC必须为S状态。(定理2的证明见附录9)

定理3. 协议满足单写多读SWMR性质,即

$$\mathcal{M}, s_0 \models AG(\forall a \in \mathcal{A}: SWMR(a, s)).$$

其中,单写多读 $SWMR(a, s)$ 定义如下:

$$\begin{aligned} SWMR(a, s) &\equiv WCount(a, s) \leq 1 \\ &\wedge (WCount(a, s) = 1 \rightarrow RCount(a, s) = 0) \\ &\wedge (RCount(a, s) \geq 1 \rightarrow WCount(a, s) = 0). \end{aligned}$$

其中, $WCount(a, s)$ 表示写者计数器,具体定义为

$$WCount(a, s) = |i \in O_{host} | C_{HCC}^i(a) = M| + \delta(C_{DDC}(a).stab = M).$$

而 $RCount(a, s)$ 表示读者计数器,具体定义为

$$RCount(a, s) = |i \in O_{host} | C_{HCC}^i(a) = S| + |j \in O_{dev} | C_{DCC}^j(a) = S|.$$

注意符号 $\delta(expr)$ 表示指示函数,如果表达式 $expr$ 为真时值为1,否则为0。(定理3的证明见附录10)

定理3指出:对于任意地址,系统在任何时刻都需满足单写多读约束。具体包括以下三点:一是同一时刻最多只允许一个写者;二是若存在写者(即处于独占修改状态M),则不允许存在任何读者(即处于共享状态S);三是如果当前存在读者,则不允许存在写者。

定理4. 协议满足数据值不变性(DVI),即

$$\mathcal{M}, s_0 \models AG(\forall a \in \mathcal{A}: DVI(a, s)), \text{ 并且}$$

$$DVI(a, s) \equiv \begin{cases} data(C_{HCC}^i(a)), & \text{若 } \exists i: C_{HCC}^i(a) = M; \\ data(C_{DCC}^j(a)), & \text{若 } \exists j: C_{DCC}^j(a) = M; \\ Mem(a), & \text{否则。} \end{cases}$$

其中, $data(\cdot)$ 表示函数从缓存行状态中提取其存储的数据值。

定理4表明:对于任何内存地址,在任何时刻,系统所能提供的该地址的“当前值”始终是正确的。换言之,如果存在一个处于独占修改M状态的缓存副本,那么该副本所持有的数据就是最新值;如果不存在这样的副本,那么主存 $Mem(a)$ 中存储的值就是最新值。(定理4的证明见附录11)

5.3 小 结

本节通过符号化的验证,严格证明了缓存一致性协议上述关键属性的正确性与完备性。在证明过程中,全局状态定义覆盖协议核心组件,动作集合与迁移函数精确描述交互逻辑,为协议验证提供理论支撑。

局限性分析:本文形式化验证在活性属性的覆盖上有所侧重,即验证无死锁性。因其是系统可运行的基本前提——若系统可能死锁,则其他属性均失去意义。然而,活性属性还包含更精细的要求,如无饥饿性和请求最终可完成性,这些尚未进行形式化证明。例如,高并发冲突场景下,某缓存控制器的请求可能因持续被抢占资源而无法获得服务。本协议虽通过优先级虚拟网络通道及冲突处理策略(见4.4节)在一定程度上降低了死锁和活锁风险,但不能彻底排除饥饿可能。此外,协议在潜在故障下的鲁棒性(如目录损坏、消息丢失或延迟异常)亦未纳入验证范围,这些故障可能导致协议进入未定义状态或违反一致性。

未来工作将聚焦于两方面:一是研究活性属性的形式化验证方法(如使用CTL/LTL描述无饥饿性并进行模型检测);二是探索容错机制的设计与验证,例如引入超时重传、目录冗余备份等技术,以增强协议在实际系统中的可靠性。

6 实验分析

本文在gem5模拟器中构建了所提出协议的缓存一致性系统,通过设计并执行一系列RubyTester测试用例,本文将所设计协议与现有协议进行了对比,从多维度验证协议是否满足预期需求。另外,模拟器主要参数配置如表9所示。

表9 gem5模拟器参数配置

处理器	X86TimingSimpleCPU
CPU主频(GHz)	1
主机/设备内存(MB)	512
缓存(KB)	4~128

6.1 数据集

实验采用斯坦福大学的公开并行测试数据集SPLASH-2(Stanford Parallel Applications for Shared Memory)进行测试。该数据集包括CHOLSKY、FFT、RADIX和LU等测试应用。表10给出了数据集对应的详细信息,其中:

表 10 仿真数据集 SPLASH-2		
数据集	数据规模	应用领域
CHOLESKY	Tk18. O	矩阵分解
FFT	256×1024 Points	快速傅里叶变换
RADIX	1048576 key, 1024 radix	基数排序算法
LU	1024×1024 matrix, 16×16 block	矩阵分解

- (1) CHOLESKY 将一个稀疏矩阵分解为一个下三角矩阵与其转置的乘积；
- (2) FFT 用于计算快速傅里叶变换；
- (3) RADIX 是基数排序算法；
- (4) LU 将一个稀疏矩阵拆分成一个下三角矩阵与一个上三角矩阵的乘积。

这些程序具有不同的数据访问模式和计算特性,能够全面地测试缓存一致性协议在不同场景下的性能表现。因此,上述测试程序被广泛应用于检验缓存一致性协议设计的正确性与有效性。

6.2 评价指标及对比方法

评价指标包括事务覆盖率与读写时延两类,具体定义如下:

(1) 事务覆盖率:该指标定义为实际测试获取的事务数量与协议中的理论事务数量的比值。仿真设计了多个测试用例,检测缓存一致性协议在不同访存路径下的事务覆盖率,以确保主机与设备在访问主机端或设备端内存时可维护缓存一致性,并能够正确地处理数据读写操作。

(2) 读写时延:仿真测试了不同场景下协议的读写时延,以评估影响协议性能表现的关键因素,例如访存路径、缓存命中率等。

此外,本文将所提出协议与经典 MESI 协议在上述并行程序集上进行了比较,重点对比不同协议完成上述应用的运行时长。通过与经典协议的比较,可以证明本文所提协议的有效性。

6.3 实验方法

针对事务覆盖率指标,通过测试不同访存路径下的事务覆盖率,确保协议的仿真模型能正确处理

各种缓存一致性场景,以提高仿真测试的置信度。

针对读写时延指标,本文首先在多种缓存命中场景下对比了不同访存路径的协议读写时延性能差异。其次,分别测试了不同缓存命中率以及缓存规模对读写请求性能的影响。

针对并行测试程序集运行时长指标,在 gem5 全系统模拟环境下,分别使用本文协议与 MESI 协议运行 CHOLESKY、FFT、RADIX 以及 LU 并行测试程序集,同时统计并对比分析运行时长。此外,实验还考察了协议的消息处理数量以及组件间的不同通信时延设置对事务执行时长的影响。

协议的实验代码与完整状态转移表获取地址:
https://github.com/chancezhw/hetero_mesi.

6.4 实验结果与分析

实验 1. 不同访存路径下事务覆盖率测试

为测试不同访存路径下事务覆盖率情况,仿真分别研究了主机/设备访问主机端内存(记为“路径一”)和主机/设备访问设备端内存(记为“路径二”)两条访存路径下的事务覆盖率,即统计实现上述访存路径中所涉及的各类事务的数量。所有仿真均在 RubyTester 环境中执行,并且需设置两个关键超参数 Checks_to_complete 以及 Wakeup_frequency。其中参数 Checks_to_complete 表示仿真中目标请求执行的总次数,仿真中设置为 10000。参数 Wakeup_frequency 表示内存访问请求触发周期。例如设置 Wakeup_frequency=2 tick,即每隔 2 tick 触发一个内存访问请求。此外,tick 表示处理器下发请求的频率,在仿真中设置 1tick=1 ns(纳秒)。另外考虑到不同 Wakeup_frequency 设置方式会影响事务覆盖率,本仿真中 Wakeup_frequency 设置为 1 tick 至 10 tick。

仿真结果如表 11 所示,其中展示了“路径一”与“路径二”条件下 HCC、HDC、DDC 及 DCC 的理论事务数量(协议设计中包含的事务总量)、测试事务数量(实际测试执行的事务总量)以及对应的覆盖率。针对路径一,DCC 覆盖率最高,达到了 88.89%,DDC 覆盖率为 62.96%;针对路径二,

表 11 不同访存路径下事务覆盖率比较						
控制器	路径一			路径二		
	理论事务数	测试事务数	覆盖率	理论事务数	测试事务数	覆盖率
HCC	54	44	81.48%	66	63	95.45%
HDC	152	110	72.37%	139	98	70.50%
DDC	54	34	62.96%	84	67	77.01%
DCC	27	24	88.89%	36	34	94.44%
总覆盖率	212/287=73.87%			262/328=79.88%		

HCC 覆盖率最高,达到了 95.45%,HDC 覆盖率为 70.50%。通过反复测试不同的随机用例,本实验能够覆盖协议中的大部分事务。具体而言,“路径一”总体覆盖率达到 73.87%,而“路径二”的总体覆盖率则达到了 79.88%。覆盖率未达到 100% 的原因分析如下。一方面,考虑到测试时长与测试精度间的平衡问题,参数 Checks_to_complete 设置可能未达到最优。理论上该参数设置越大越好,但是随着该参数的增大,测试时长同步增加,即覆盖率提升空间随该参数的增加而逐渐递减。另一方面,经检查,这些未覆盖的事务在测试用例中触发条件较为严苛,属于低频事务。例如,一致性组件 A 在 ES_S 状态下处理读监听无效 MemRdSnPInv 请求是一个低频事务。该事务的触发须满足严苛的时序竞争条件:组件 B 的 GetS 请求必须与组件 C 的 GetM 请求,在组件 A 完成状态

E 向 S 转换前的极短窗口内几乎同时抵达,因此实际发生概率较低。

综上所述,不同访存路径下事务整体覆盖率接近 80%。此外,注意到涉及事务数量最多的两个控制器分别为 HDC 与 DDC,这体现了它们在本协议中的重要性,并且仿真结果表明其覆盖率也达到 70% 以上,满足协议设计的预期。

实验 2. 不同缓存命中场景对读写时延性能影响

为验证不同缓存命中场景对读写时延性能影响,仿真分别测试了主机访问主机端内存(记为“路径三”)以及主机访问设备端内存(记为“路径四”)两条访存路径下的协议读写时延性能。针对上述两条路径,仿真分别比较了协议完成 Load 读请求、Store 写请求及触发 Replacement 的时延情况。仿真结果见表 12 与表 13。

表 12 路径三条件下 Load、Store 以及 Replacement 请求完成时延比较

实验场景	场景分类	请求访存平均时延(ns)		
		Load	Store	Replacement
HCC 命中(缓存状态为 M)	HCC 缓存命中	1	1	68
HCC 命中(缓存状态为 E)		1	1	10
HCC 命中(缓存状态为 S)		1	62	10
HCC 未命中,HDC 命中(缓存状态为 M)	HDC 目录命中	16	16	
HCC 未命中,HDC 命中(缓存状态为 E)		16	16	-
HCC 未命中,HDC 命中(缓存状态为 S)★		52	62	
HCC 与 HDC 未命中,DDC 命中(缓存状态为 M)☆	HDC 目录未命中	31	43	
HCC 与 HDC 未命中,DDC 命中(缓存状态为 E/S)		72	72	-
HCC 与 HDC 未命中,DDC 未命中(缓存状态为 I)		76	76	

表 13 路径四条件下 Load、Store 以及 Replacement 请求完成时延比较

实验场景	场景分类	请求访存平均时延(ns)		
		Load	Store	Replacement
HCC 命中(缓存状态为 M)	HCC 缓存命中	1	1	73
HCC 命中(缓存状态为 E)		1	1	18
HCC 命中(缓存状态为 S)		1	-	18
HCC 未命中,HDC 命中(缓存状态为 M)	HDC 目录命中	16	16	
HCC 未命中,HDC 命中(缓存状态为 E)		16	16	-
HCC 未命中,HDC 命中(缓存状态为 S)		62	72	
HCC 与 HDC 未命中,DDC 命中(缓存状态为 M)	HDC 目录未命中	31	31	
HCC 与 HDC 未命中,DDC 命中(缓存状态为 E/S)		62	62	-
HCC 与 HDC 未命中,DDC 未命中(缓存状态为 I)		77	77	

在仿真过程中,Load 与 Store 请求的通常执行过程如下。首先,HCC 检查是否命中。其次,若 HCC

未命中,则会进一步检查 HDC 是否命中。最后,若 HDC 未命中,则会检查 DDC/DCC 是否命中。因

此,请求时延随访问层级加深而增加。表12统计了“路径三”对应的Load、Store以及Replacement请求的时延性能。

从仿真结果可以看出,Load与Store请求时延性能均表现出以下特征:当HCC缓存命中时,内存访问请求只需直接访问缓存即可,因此时延最低且为1 ns;当HDC目录控制器命中且为M/E状态时,内存访问请求可以直接从主机缓存或者设备缓存中获取数据,无需访问内存,仿真结果表明时延为16 ns;当HDC目录控制器未命中且处于E/S/I状态时,获取数据时需要真正访问内存,因此时延最大且超过70 ns。另外,注意到Case★与Case☆的时延性能未表现出上述特征。这是因为Case★中请求指令过程必须访问内存,且等待内存操作完成后才能进行后续指令处理,因此时延较大。反观Case☆,其执行流程未涉及内存操作,因而耗时较短。横向来看,Load与Store请求时延性能在各场景分类中表现相当。这是因为Load与Store请求在本协议中执行的状态转换过程(状态转换路径长度)基本一致。注意到在HCC命中且状态为S的情况下,Store请

求执行过程中需真正访问内存,因而其耗时较长。另外,需要指出的是,只有当HCC命中时,才会触发Replacement请求,且当缓存块为E或S时,执行PutE/PutS命令无需写回内存,因此时延短且为10 ns;而缓存块为M时执行PutM命令需写回内存且耗时68 ns,因此时延开销大。

综上所述,涉及真正访问内存的请求执行时延较大,而仅需访问缓存的请求时延相对较低。这体现了缓存一致性协议设计的重要性与有效性。同理,表13统计了“路径四”对应的Load、Store以及Replacement请求的时延性能。仿真结果与“路径三”表现出类似的性质。

此外,本仿真在类似架构下运行了经典的MESI协议,并且分别在“路径三”与“路径四”访存路径下测试协议读写时延性能。仿真结果如表14所示。仿真结论包括以下三点:一是MESI协议的读写时延性能表现出与本文提出协议的读写时延性能变化趋势保持一致;二是MESI协议支持的测试场景种类少于本协议;三是在相同测试场景下,本协议的读写时延性能与MESI协议性能相当。

表 14 MESI协议的Load、Store以及Replacement请求完成时延比较

访存路径	实验场景	请求访存平均时延(ns)		
		Load	Store	Replacement
路径三	缓存命中(缓存状态为M)	1	1	63
	缓存命中(缓存状态为E)	1	1	10
	缓存命中(缓存状态为S)	1	1	10
	缓存未命中,目录命中(缓存状态为M)	16	16	
	缓存未命中,目录命中(缓存状态为E)	16	16	
	缓存未命中,目录命中(缓存状态为S)	63	65	-
	缓存未命中,目录未命中(缓存状态为I)	63	63	
路径四	缓存命中(缓存状态为M)	1	1	65
	缓存命中(缓存状态为E)	1	1	11
	缓存命中(缓存状态为S)	1	1	11
	缓存未命中,目录命中(缓存状态为M)	21	21	
	缓存未命中,目录命中(缓存状态为E)	21	21	
	缓存未命中,目录命中(缓存状态为S)	68	70	-
	缓存未命中,目录未命中(缓存状态为I)	68	68	

实验3. 不同缓存命中率对读写时延性能影响
本仿真在固定“路径四”且缓存规模设置为64 KB条件下,分别观测缓存命中率由0~1变化过程中对Load请求与Store请求的时延性能变化趋势。仿真统计了本文提出协议的Load请求(记为

“P-Load”)、Store请求(记为“P-Store”)以及MESI协议的Load请求(记为“MESI-Load”)、Store请求(记为“MESI-Store”)执行的平均时延。仿真结果如图12所示。

从仿真结果的整体变化趋势上来看,缓存命中

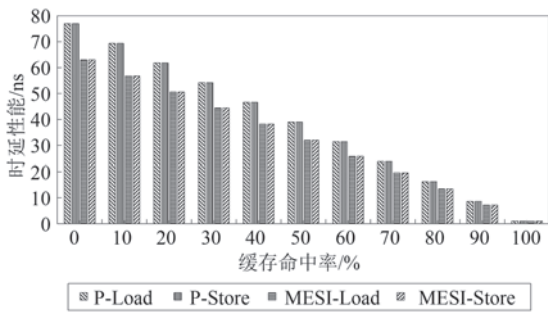


图12 不同缓存命中率下各请求时延对比

率越高,读请求与写请求执行的耗时越低,即时延越低。上述仿真结果符合预期的理论分析。一方面,较高的缓存命中率意味着所需数据已被存储于缓存中,可被快速获取和使用。另一方面,较低的缓存命中率意味着所需数据大概率未被存储于缓存中,需从内存中搬运至缓存后,才能后续使用。

具体而言,当缓存命中率为0时,意味着请求数据需真正访问内存,因此其平均耗时最长,P-Load/Store与MESI-Load/Store均超过60 ns;当缓存命中率为1时,此时缓存中有请求数据,因此直接返回相应数据,而无需访问内存,P-Load/Store与MESI-Load/Store耗时仅为1 ns。另一方面,注意到P-Load与P-Store请求的执行时长性能相当。主要原因是Load请求执行的GetS指令与Store请求执行的GetM指令,两者在状态转化过程中的路径长度差异不大,导致它们所花费的时间开销基本一致。

实验4. 不同缓存规模对读写时延性能影响

本仿真在“路径四”条件下,测试缓存规模由4 KB增加至64 KB过程中如何影响所提出协议的P-Load与P-Store请求的时延性能以及MESI协议的MESI-Load请求与MESI-Store请求的时延性能。图13统计了上述两类缓存一致性协议请求的平均时延结果。

从仿真数据中可以看出,上述协议各自请求的整体时延随着缓存规模的增大而逐渐降低。具体而

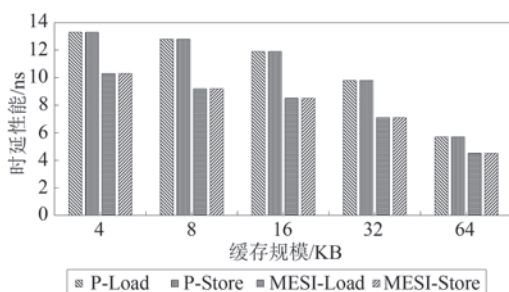


图13 不同缓存规模下各请求时延对比

言,当缓存规模为4KB时,本协议的P-Load与P-Store请求时延约为13.3 ns,而MESI协议的MESI-Load请求与MESI-Store请求时延约为10.3 ns。当缓存规模为64 KB时,本协议的P-Load与P-Store请求时延约为5.7 ns,而MESI协议的MESI-Load与MESI-Store请求时延约为4.5 ns。缓存由4 KB增加至64 KB时,本协议与MESI协议的请求平均时延降幅分别为57.14%与56.31%。产生上述仿真结果的原因分析如下:当缓存规模较小时,所需数据大概率未被存储于缓存中,可能需访问内存获取或写回数据,导致该条件下请求时延较大;当缓存规模较大时,所需数据大概率被存储于缓存中,无需访问内存获取并写回数据或缓存写回内存的频率会降低,导致该条件下请求时延较小。另一方面,与本协议相比,MESI协议本身复杂度不高,即瞬态数量相对较少,因而造成状态转换过程中请求时延更短。仿真数据表明,缓存由4 KB增加至64 KB过程中,MESI协议请求时延比本文所提出的协议少21.05%~28.57%。

实验5. 不同并行程序测试集的协议运行时长对比

为评估协议在实际应用数据集上的运行时长,本文在SPLASH-2的CHOLESKY、FFT、RADIX和LU数据集上进行了仿真,并与经典的MESI协议进行对比。仿真结果如图14所示。

本仿真分别测试缓存容量为32 KB、64 KB以及128 KB三种情况,仿真结果如图14所示。图中横轴表示不同缓存容量,纵轴表示测试程序运行时长比。具体地,为方便直观比较,本文将缓存规模为32 KB且在FFT数据集上使用MESI协议所获取的运行时长作为基准,而运行时长比定义为与基准运行时长的比值。该比值越大,表明测试程序运行时长越长;比值越小,则运行时长越短。

相较于经典的MESI协议,本协议支持异构计算单元间的数据交互,并且维护缓存一致性。仿真

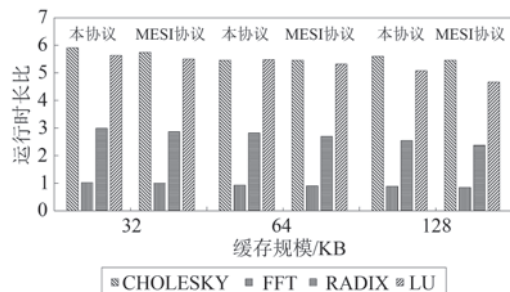


图14 不同数据集上各协议的运行时长对比

结果表明在相同缓存设置条件下,本协议运行时长仅比MESI协议在同一测试程序运行时多2%~4%。例如,缓存规模设置为64 KB时,基于本文设计的协议运行CHOLESKY、FFT、RADIX和LU测试程序的运行时长比依次为5.91、1.02、2.99、5.63,而基于MESI协议运行四个测试程序的运行时长比依次为5.74、1.0、2.87、5.5。仿真结果验证了基于统一状态表示的缓存一致性协议与经典MESI协议在实际应用中运行时长性能相当,证明了所提出协议的有效性。另一方面,与缓存容量为32 KB的仿真结果相比,缓存容量为64 KB以及128 KB设置下,基于两种不同协议运行的四个测试程序运行时长均表现出降低的情况。例如,MESI协议在FFT数据集运行时长比由1.0(32 KB)降低至0.85(128 KB),本文提出协议在CHOLESKY数据集上运行时长比由5.91(32 KB)降低至5.16(64 KB)。然而,与缓存容量为64 KB的仿真结果相比,缓存容量为128 KB的仿真性能并未显著提升。因此在缓存一致性协议的实际设计中,缓存容量不宜过大。

实验6. 不同协议处理一致性消息数量对比

本实验在“主机端访问设备端内存的访存路径”条件下,以1 ns为间隔连续下发1000次Store(写请求)指令,分别统计并对比了本文提出的协议与MESI协议的目录控制器所处理的一致性消息数量。具体而言,当主机端处理器下发Store写请求以访问设备端内存时,目录控制器所处理的一致性消息主要包括读监听无效(MemRdSnPInv)、写监听无效(MemWrSnP)、数据响应(MemData)以及响应确认(MemAck)四类事务。图15分别展示了本文协议与MESI协议在上述几类一致性消息数量上的统计结果。

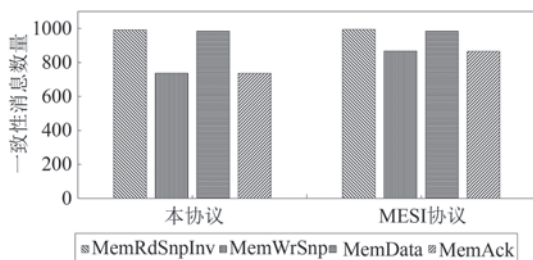


图15 不同协议的一致性消息数量对比

实验结果表明,两种协议在处理读监听无效(MemRdSnPInv)与数据响应(MemData)消息的数量上基本相当。然而,在写监听无效消息方面,本文

协议的数量为737条,相较于MESI协议的867条减少了14.99%;在响应确认消息方面,本文协议的数量为736条,较MESI协议的866条降低了15.01%。总体来看,本文协议的一致性消息总数降低了7.06%。以上结果验证了本文协议在处理请求消息时具有更高的工作效率。

实验7. 不同交互时延对请求处理性能的影响

本实验研究了两种不同交互路径时延参数设置对请求处理性能的影响:①主机端目录控制器(HDC)与设备端目录控制器(DDC)之间的时延变化(由1 ns增加至32 ns);②主机端目录控制器(HDC)与主机端缓存控制器(HCC)之间的时延变化(由1 ns增加至32 ns)。实验在“主机端访问设备端内存”的访存路径下,以1 ns为间隔连续下发1000次Store(写请求)指令,分别统计在HDC-DDC与HDC-HCC之间时延由1 ns逐步增加至32 ns的过程中,完成全部Store请求所需的总时长。实验结果如图16所示。

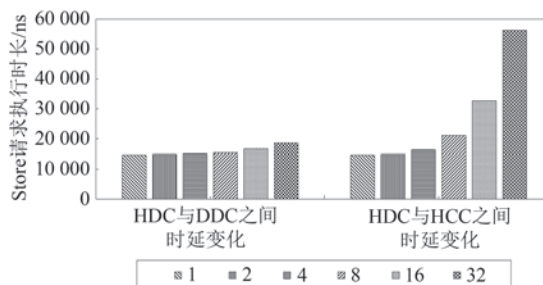


图16 不同交互时延对请求处理性能的影响

实验结果表明,在上述两种时延设置下,请求执行总耗时均随时延增大而增长,但增长幅度存在显著差异。在HDC-DDC时延变化(方式①)中,当时延从1 ns增至32 ns时,总耗时由14738 ns上升至18627 ns,增幅约为26.39%,表明其性能受时延影响相对较小。而在HDC-HCC时延变化(方式②)中,同等条件下总耗时从14738 ns增加至56306 ns,增幅达282%,表明该路径对时延变化极为敏感。上述结果揭示了不同控制器间时延对系统性能的影响特性:HDC与DDC间的交互表现出较好的时延容忍性;而HDC与HCC间的交互则具有较高的时延敏感性,其通信效率更易受路径时延的制约。

7 相关工作

异构计算系统致力于将多种计算单元集成到一个共享内存系统中,并确保其行为符合一致性模型所规定的内存访问顺序。近年来,国内外学术界与产业界学者广泛开展了相关技术研究,并取得阶段性突破。以下分别从协议设计、形式化验证以及两者相结合的研究工作等方面展开介绍。

7.1 缓存一致性协议研究进展

在产业界,处理缓存一致性的主要途径是开发协议标准。例如,AMD、IBM等行业巨头于2016年联合推出了CCIX^[20]、Gen-Z^[21]和OpenCAPI^[22]三种新型互连标准。CCIX旨在通过统一的缓存一致性机制连接异构计算单元,以解决传统PCIe接口缺乏缓存一致性的问题,提升数据共享效率与系统性能,为数据中心等高带宽低时延场景提供互连方案。其优势在于通过硬件级协议避免软件层数据同步开销,从而简化编程模型。研究表明,该方案读取操作时延较PCIe降低33%,但写入操作因一致性开销增加了约32%的时延^[23]。Gen-Z以内存为中心,采用基于数据包的通信协议和共享内存语义,实现对全局内存空间的直接访问。其优势在于良好的可扩展性,支持数千节点规模,但要求设备端集成复杂的协议栈和缓存一致性代理,增加了芯片设计成本。OpenCAPI是一种高性能、低时延的开放互连标准,具备高带宽和灵活的拓扑结构,尤其适用异构加速场景。然而受限于产业生态支持不足,目前仅应用于IBM Power体系及特定高性能计算领域。此外,Intel、AMD等公司于2019年推出了CXL协议标准^[24],致力于解决异构计算环境中缓存一致性、内存带宽不足和资源利用率低等问题。该协议在PCIe规范之上制定CXL.cache/memory/io三类子协议,核心优势在于高带宽和良好的生态兼容。与前述协议相比,CXL凭借Intel主导的产业联盟已成为异构互连的事实标准,但其产业化相对滞后。

在学术界,Nagarajan等人简要讨论了异构一致性模型,并引入了复合一致性模型的概念,但未提供融合两类协议的一般方法^[25]。Hower等人提出了HRF内存模型用于处理不同作用域的同步操作,但未解决多种异构协议的组合问题^[26]。Iorga等人首次从公理化和操作化角度正式定义了异构计算系统的内存模型,并实施了相应验证^[27]。此外,Crossing Guard提供了一种安全、标准化的缓存一致性接口,

解决了加速器设计复杂性与主机系统安全性之间的矛盾,其优势在于能在保持高性能的同时,为异构计算系统提供可靠性保障^[28]。

近期,研究者采用层次化协议转换与翻译方案以应对系统缓存一致性维护问题。例如,斯坦福大学Rechel等人提出了一种基于监听更新策略的MEMGLUE协议^[11],利用异构集群松散排序行为,实现了多种内存一致性模型的有效兼容。该协议包括翻译器与一致性控制器两层架构:翻译器基于C11内存一致性模型(该模型旨在为多线程并发编程提供标准化的内存访问规则,解决多核环境下缓存一致性、指令重排及数据竞争等问题)将集群内部内存一致性模型转换为统一的MEMGLUE协议;一致性控制器作为协议管理组件,负责接收各翻译器发送的消息,并对其进行统一重排序与路由调度,最终实现集群间异构内存一致性模型的兼容。此外,爱丁堡大学Nicolai等人提出了基于目录策略的异构集群复合一致性协议融合方法HeteroGen^[13]。该方法通过代理缓存技术融合不同集群的一致性模型,在遵循全局复合一致性模型的同时保留各集群的一致性模型。作者还设计了针对复合一致性模型的litmus测试方案,并使用形式化检测工具验证协议是否满足预期的复合一致性。然而,上述协议的验证仅仅测试了少量特殊的测试样例,可能无法覆盖所有边界情况和异常行为。

讨论:在协议架构层面,MEMGLUE协议针对异构设备集群间采用不同一致性协议的问题,设计了基于局部转换与全局一致性控制器的双层架构,以实现系统范围的一致性维护。然而,该设计不可避免地引入了协议转换与通信开销。此外,在协议实现机制上,MEMGLUE采用全局消息广播的方式,导致其系统可扩展性受到限制。HeteroGen则采用目录融合机制,将不同协议整合为统一的复合型协议。该机制为每个集群设立独立的代理缓存,当集群A需访问集群B的数据时,代理缓存模拟集群B的缓存控制器向对应目录发起请求,从而使集群A将集群B视为其子缓存,反之亦然。尽管HeteroGen通过目录结构有效降低了MEMGLUE中的广播开销,但其协议融合过程依赖于特定的转换框架与工具,因而在所能支持的协议类型上存在局限。CXL.cache通过维护一个全局目录来动态跟踪所有外部设备的缓存行状态,但随着设备数量与缓存容量的增加,目录元数据开销与监听广播流量可能构成可扩展性瓶颈。

与上述方案不同,本文提出的一致性协议面向异构计算系统,设计了包括HCC、HDC、DDC与DCC在内的四个核心组件,构建了主机端与设备端协同的分布式目录结构。该协议将MESI协议引入异构计算,并为各组件定制了事务类型与状态迁移规则,在确保一致性的同时支持多样化的访存路径。此外,该协议采用统一的状态表示方法,从而避免了协议转换与翻译带来的开销与复杂性问题。

7.2 形式化验证研究进展

协议的验证主要包括模拟验证和形式化验证两大类。模拟验证通过在系统输入端施加激励信号,并观察输出是否与设计意图一致来判断正确性,该方法简单有效,但仅能证明协议设计存在错误,而无法证明其正确性,且验证覆盖率较低,导致结果可靠性有限。形式化验证^[29]则使用形式化语义的符号、公式以及定理来描述带参协议的设计规范和属性,运用数学方法验证协议是否满足正确性、可靠性、无死锁等性质,具有完备性,能够严格判定设计的正确性,已成为主流验证技术,主要包括基于定理证明和模型检测技术的验证方法。

(1)基于定理证明技术的验证方法使用数学逻辑公式描述协议,将其转化为逻辑模型,将系统性质表述为待证明的定理,进而运用数学定理与公式进行推导和证明。该方法本质上是从带参协议的系统公理中归纳证明系统性质的过程,亦称归纳证明法。基于该方法开发的定理证明工具包括英国剑桥大学与德国慕尼黑技术大学联合开发的基于高阶逻辑的交互式通用定理证明器Isabelle^[30],以及法国国家信息研究所开发的Coq^[31]。这些工具已被国内外学者和机构采用。近年来,北京大学詹乃军教授团队在Coq定理证明器中验证了操作系统内核权限访问控制的正确性^[32]。北京航空航天大学章乐平等基于Isabelle形式化了L4虚拟内存子系统的功能正确性、功能安全和信息安全三方面属性,并证明了所提形式模型满足上述属性^[33]。综上所述,定理证明方法基于公理与推理规则进行数学证明,保证了验证过程的严谨性,可覆盖所有可能的执行路径和输入条件。然而,尽管自动化证明工具可辅助部分证明,复杂系统的验证仍需人为干预,自动化程度有限。

(2)基于模型检测技术的验证方法通过自动搜索状态空间,判断性质是否在所有可达状态都成立;若不成立,则终止搜索并输出反例,设计人员可据此分析错误原因并修正协议设计^[34]。该方法依赖于

SPIN^[15]、UPPAAL^[16]、NuSMV^[35]等模型检测工具。例如,Dill等人使用Murphi验证了Sun可扩展共享存储器系统中三个节点间的带参协议^[36]。McMillan等人通过NuSMV验证了带参缓存一致性协议^[37]。SPIN已被广泛应用于总线协议、局域网控制器及安全协议等的形式化验证。此外,中国科学院软件所林惠民院士等人开发了一款基于一阶逻辑的模型检测工具,并对典型的GERMAN2004协议进行了验证,发现了该协议在数据一致性方面存在的问题^[38]。近期,西安电子科技大学段振华团队开发了一款支持线程迁移依赖性细化的并发程序模型检测工具CDG4CPV^[39],实验表明该方法可将探索状态数平均降低91.38%。南京航空航天大学黄志球团队提出了基于交叉熵迭代的统计模型检测方法^[40],用于解决随机混成系统稀有属性采样问题。综上所述,该类方法自动化程度较高,但主要适用于状态空间有限的带参协议,且随着协议规模增大,状态空间爆炸问题日益显著。为此,研究者提出了抽象与对称等优化技术以缩减待验证状态空间^[17,41]。

尽管面向多核处理器的缓存一致性协议研究已开展多年,但针对异构计算系统的相关研究相对较少,其缓存一致性协议的形式化验证亦不够完善,有待进一步探索。

7.3 缓存一致性协议的形式化验证研究进展

缓存一致性协议的形式化验证是确保多核系统可靠性的关键,但面临状态空间爆炸、跨层交互复杂、验证效率低下等诸多挑战。近期研究从不同层次提出了创新方法。

在模块化组合验证方面,针对异构内存一致性模型验证缺乏模块化支持的问题,RealityCheck^[17]形式化验证工具应运而生。该工具支持采用层次化模块描述微架构顺序约束,并通过接口抽象实现各模块的独立验证与可扩展的组合验证。实验对TSO(Total Store Order)与RVWMO(RISC-V Weak Memory Order)模型进行了验证,测试用例结果表明,抽象优化最高可降低32%的验证时间。该方法优势在于贴合实际设计流程,提升了验证的可扩展性;不足之处在于仍属于有界验证,无法覆盖所有程序,且在约束较强时接口验证耗时较长。此外,针对TileLink协议模块化规范中全局内存排序的集体语义难以本地验证的问题,文献[42]提出基于参考对象与循环假设的形式化推理方法。该工作利用Ivy语言构建模块化、可扩展的规范,并结合组件验证与模块化约束随机测试进行验证。实验结果表明,该

方法成功验证了缓存、目录等组件的合规性,修正了25处规范错误。该工作的不足之处在于组件验证需依赖大量人工辅助的不变式与特定的工具链。因此,复杂组件验证成本较高。

在协议层与互连层的交叉验证方面,研究学者提出了集成验证工具 ADVOCAT^[43],通过融合 I/O 状态自动机与 XMAS 模型以支持跨层建模,并自动推导协议状态与在途数据包间的关联不变量,实现跨层死锁的自动检测与验证。

在验证效率优化方面,针对 CXL. cache 协议规范中存在的模糊与冗余问题,文献[44]采用 Isabelle 证明助手构建其形式化模型,识别出5处缺陷并提出修复方案。该工作通过场景验证与机械化证明确保模型有效性,证明其满足单写多读特性,并且开发了 auto_sketch 工具以提升大规模证明的自动化效率。

针对缓存一致性协议回归验证中测试集庞大、执行耗时长的问题,文献[45]提出一种以缺陷检测率与功能覆盖率为目标的多目标测试用例优先级排序算法。该方法通过测试子集与子集内用例的两级排序及轮询选取生成优化序列。在64核处理器上的实验表明,仅执行40%的用例即可检测全部缺陷,80%的用例能达到近100%的功能覆盖率。

文献[46]将同地址顺序性作为定理,基于 RVWMO 现有公理与规则,将同地址访存序列抽象为确定有限状态自动机,并通过归纳法证明了12种基本序列模式及13种模式转换的正确性,最终使用形式化验证工具 Coq 完成验证。实验表明验证效率提升超过400%,且能简化 Litmus 测试的合法性判定。但该证明尚未覆盖所有序列模式,在复杂场景下的扩展性仍有不足。

此外,在模拟验证与形式化验证的结合方面,近期研究^[47]针对 gem5 模拟器中缓存一致性协议设计与验证流程割裂的问题,提出了自动化模型检验工具 Rubyphi。该方法能够自动提取 gem5 中 Ruby 系统描述的协议逻辑,并将其转换为 Murphi 模型检验语言,从而实现了从模拟器到形式化验证的无缝衔接。实验表明 Rubyphi 可有效完成协议建模与验证,成功检测出 gem5 中 MI 协议的两个设计错误,显著降低了建模门槛与时间成本。但其仍面临状态空间爆炸问题,且支持的外部事务类型有限,在可扩展性与事务覆盖范围方面有待进一步优化。

综上所述,上述工作共同推动了协议的形式化验证向更实用、可扩展、与设计流程紧密结合的方向

发展,但各自在覆盖范围、自动化程度上仍存在一定局限。

8 结 论

本文提出一种统一状态表示的缓存一致性协议,以解决异构系统面临的协议设计与验证挑战。首先,设计了涵盖 HCC、HDC、DDC、DCC 四个核心组件的一致性维护架构,并概述了一致性维护过程涉及的状态定义、事务、网络通道等概念。其次,重点阐述了目录控制器 HDC 与 DDC 结构设计与工作原理。针对不同访存路径,研究了读写请求事务流程处理过程中一致性维护核心组件的状态转换关系。此外,采用抽象的公理化形式验证方法,对协议的无死锁性、单写多读及数值不变性等属性实施验证。为评估协议有效性,在 gem5 仿真器环境中制定事务覆盖、读写时延等评价指标,并与经典 MESI 协议进行了对比仿真。仿真结果表明提出的协议在并行测试集上与经典 MESI 协议的运行时长性能相当(差距仅介于2%~4%)。

此外,目前验证工作主要依靠定理证明与 gem5 仿真,两者相对独立。未来我们将探索基于模型检测与 gem5 相结合的协同验证策略:一方面,利用模型检测工具(如 SPIN、Murphi)自动搜索协议状态空间,验证活性属性及复杂并发场景下的正确性,弥补定理证明在自动化方面的不足;另一方面,将模型检测发现的反例或边界条件转化为 gem5 仿真测试用例,进行动态性能评估与验证。这种“形式化+仿真”的双重验证方法有望进一步提升协议设计的可靠性和效率,同时为协议优化提供更全面的指导。

致 谢 感谢国家重点研发计划项目(2024YFE0210800)资助。

参 考 文 献

- [1] Shen S, Yao Z W, Li C Y, et al. Scaling vision-language models with sparse mixture of experts//Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics. Toronto, Canada, 2023: 11329-11344
- [2] Chen Y M, Liu S L, Chen Y J, et al. A scheduling algorithm for heterogeneous computing systems by edge cover queue. Knowledge-Based Systems. 2023, (265): 110369
- [3] Wang X, Huang L N, Zhu Y X, et al. Addressing memory wall problem of graph computation in reconfigurable system//

- Proceedings of the 17th IEEE International Conference on High Performance Computing and Communications. New York, USA, 2015: 302-307
- [4] Chen Zhi-Qiang, Zhou Hong-Wei, Feng Quan-You, et al. Design and implementation of configurable cache coherence protocol for multicore processor. *Journal of Computer Research and Development*. 2021, 58(6): 1166-1175 (in Chinese)
(陈志强, 周宏伟, 冯权友等. 面向多核处理器的可配置缓存一致性协议设计与实现. *计算机研究与发展*. 2021, 58(6): 1166-1175)
- [5] Menezes L G, Puente V, Gregorio J A. Flask coherence: a morphable hybrid coherence protocol to balance energy, performance and scalability//*Proceedings of the 21st IEEE International Symposium on High Performance Computer Architecture*. San Francisco, USA, 2015: 198-209
- [6] Wilkins M, Westrick S, Kandiah V, et al. Warden: specializing cache coherence for high-level parallel languages//*Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization*. Montreal, Canada, 2023: 122-135
- [7] Zhu M C, Shahab A, Katsarakis A, et al. Invalidate or update? revisiting coherence for tomorrow's cache hierarchies//*Proceedings of the 30th International Conference on Parallel Architectures and Compilation Techniques*. Atlanta, USA, 2021: 226-241
- [8] Egger S J, Katz R H. Evaluating the performance of four snooping cache eggers coherency protocol. *ACM SIGARCH Computer Architecture News*. 1989, 17(3): 2-15
- [9] Moshovos A. Region scout: exploiting coarse grain sharing in snoop-based coherence//*Proceedings of the 32nd International Symposium on Computer Architecture*. Madison, USA, 2005: 234-245
- [10] Sorin D, Hill M, Wood D. A primer on memory consistency and cache coherence. Cham, Switzerland: Morgan & Claypool Publishers, 2011
- [11] Cleaveland R, Trippel C. Memory consistency model-aware cache coherence for heterogeneous hardware//*Proceedings of the 24th Conference on Formal Methods in Computer-Aided Design*. Trondheim, Norway, 2024: 163-174
- [12] Duan Zhuo-Hui, Kiu Hai-Kun, Zhao Jin-Wei, et al. A reconfigurable cache consistency mechanism for distributed memory pool. *Journal of Computer Research and Development*. 2023, 60(9): 1960-1972 (in Chinese)
(段卓辉, 刘海坤, 赵金玮等. 一种可动态配置的分布式内存池缓存—致性机制. *计算机研究与发展*. 2023, 60(9): 1960-1972)
- [13] Oswald N, Nagarajan V, Sorin D J, et al. Heterogen: automatic synthesis of heterogeneous cache coherence protocols//*Proceedings of the 28th IEEE International Symposium on High-Performance Computer Architecture*. Seoul, Republic of Korea, 2022: 756-771
- [14] He Xi-Ming, Ma Sheng, Huang Li-Bo, et al. A simple and efficient cache coherence protocol based on self-updating. *Journal of Computer Research and Development*. 2019, 56(4): 719-729 (in Chinese)
(何锡明, 马胜, 黄立波等. 一种基于自更新的简单高效 Cache—致性协议. *计算机研究与发展*. 2019, 56(4): 719-729)
- [15] Zheng Y, Ru K, Feng X, et al. The overview of spin in software model checking//*Proceedings of the 6th International Conference on Data Science in Cyberspace*. Melbourne, Australia, 2021: 654-661
- [16] Bengtsson J, Larsen K, Larsson F, et al. Uppaal: a tool suite for automatic verification of real-time systems. Berlin, Germany: Springer Berlin Heidelberg, 1996
- [17] Manerkar Y A, Lustig D, Martonosi M. Realitycheck: bringing modularity, hierarchy, and abstraction to automated microarchitectural memory consistency verification. *arXiv preprint arXiv: 2003.04892*, 2020
- [18] AL-Waisi Z, Agyeman M O. An overview of on-chip cache coherence protocols//*Proceedings of the 4th IEEE Intelligent Systems Conference*. London, UK, 2017: 304-309
- [19] Caheny P, Alvarez L, Valero M, et al. Runtime-assisted cache coherence deactivation in task parallel programs//*Proceedings of the 30th International Conference for High Performance Computing, Networking, Storage and Analysis*. Dallas, USA, 2018: 454-465
- [20] Koenen D, Defilippi J. Ccix: a new coherent multichip interconnect for accelerated use cases http://www.armtechforum.com.cn/attached/article/C7_CCIX20171226161955.pdf
- [21] Hong S, Kwon W O, Oh M H. Hardware implementation and analysis of gen-z protocol for memory-centric architecture. *IEEE Access*. 2020, 8: 127244-127253
- [22] Stuecheli J, Starke W J, Irish J D, et al. Ibm power9 opens up a new era of acceleration enablement: opencapi. *IBM Journal of Research and Development*. 2018, 62(4/5): 8-1
- [23] Tamimi S, Stock F, Koch A, et al. An evaluation of using ccix for cache-coherent host-fpga interfacing//*Proceedings of the 30th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*. New York, USA, 2022: 1-9
- [24] Das S D, Blankenship R, Berger D. An introduction to the compute express link (cxl) interconnect. *ACM Computing Surveys*. 2024, 56(11): 1-37
- [25] Nagarajan V, Sorin D J, Hill M D, et al. A primer on memory consistency and cache coherence, 2nd edition. *Synthesis Lectures on Computer Architecture*. Cham, Switzerland: Springer Nature, 2020
- [26] Hower D R, Hechtman B A, Beckmann B M, et al. Heterogeneous-race-free memory models//*Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*. Salt Lake City, USA, 2014: 427-440
- [27] Iorga D, Donaldson A F, Sorensen T, et al. The semantics of shared memory in intel cpu/fpga systems. *Proceedings of the ACM on Programming Languages*. 2021, 5(120): 1-28
- [28] Olson L E, Hill M D, Wood D A. Crossing guard: mediating host-accelerator coherence interactions. *ACM SIGARCH Computer Architecture News*. 2017, 45(1): 163-176
- [29] Wang Ji, Zhan Nai-Jun, Feng Xin-Yu, et al. Overview of formal methods. *Journal of Software*. 2019, 30(1): 33-61 (in Chinese)
(王戟, 詹乃军, 冯新宇等. 形式化方法概貌. *软件学报*. 2019,

- 30(1): 33-61)
- [30] Huffman B, Kuncar O. Lifting and transfer: a modular design for quotients in Isabelle/HOL//Proceedings of the 3rd International Conference on Certified Programs and Proofs. Melbourne, Australia, 2013: 131-146
- [31] Benzmüller C, Woltzenlogel P B. Interacting with modal logics in the Coq proof assistant//Proceedings of the 10th International Computer Science Symposium. Listvyanka, Russia, 2015: 398-411
- [32] Xu Jia-Le, Wang Shu-Ling, Li Li-Ming, et al. Formal verification of capability-based access control in operating system kernel. *Journal of Software*. 2025, 36(8): 3570-3586 (in Chinese)
(徐家乐, 王淑灵, 李黎明等. 操作系统内核能力访问控制的形式验证. *软件学报*. 2025, 36(8): 3570-3586)
- [33] Zhang Le-Ping, Zhao Yong-Wang, Wang Bu-Yang, et al. Formal verification of virtual memory subsystem in L4. *Journal of Software*. 2023, 34(8): 3527-3548 (in Chinese)
(章乐平, 赵永望, 王布阳等. L4 虚拟内存子系统的形式化验证. *软件学报*. 2023, 34(8): 3527-3548)
- [34] Manerkar Y A, Lustig D, Martonosi M, et al. Rtlcheck: verifying the memory consistency of RTL designs//Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture. Boston, USA, 2017: 463-476
- [35] Kova M, Bentahar J, Maamar Z, et al. A formal verification approach of conversations in composite web services using nsmv. *New Trends in Software Methodologies, Tools and Techniques*. 2013, 199(1): 245-261
- [36] Dill D L, Drexler A J, Hu A J, et al. Protocol verification as a hardware design aid//Proceedings of the 10th International Conference on Computer Design: VLSI in Computers and Processors. Cambridge, USA, 1992: 522-525
- [37] McMillan K L, Schwalbe J. Formal verification of the Encore Gigamax cache consistency//Proceedings of the 3rd International Symposium on Shared Memory Multiprocessor. Sapporo, Japan, 1991: 1-5
- [38] Pan H, Lin H M, Lv Y. Model checking data consistency for cache coherence protocols. *Journal of Computer Science and Technology*. 2006, 21(5): 765-775
- [39] Su Jie, Yang Zu-Chao, Tian Cong, et al. Model checking tool for concurrent program based on constrained dependency graph. *Journal of Software*. 2023, 34(7): 3067-3079 (in Chinese)
(苏杰, 杨祖超, 田聪等. 基于约束依赖图的并发程序模型检测工具. *软件学报*. 2023, 34(7): 3067-3079)
- [40] Fang Bing-Wu, Huang Zhi-Qiu, Xie Jian. Statistical model checking for verification of rare properties of stochastic hybrid system. *Journal of Software*. 2022, 33(10): 3717-3731 (in Chinese)
(房丙午, 黄志球, 谢健. 随机混成系统稀有属性的统计模型检测方法. *软件学报*. 2022, 33(10): 3717-3731)
- [41] Delzanno G. Constraint-based verification of parameterized cache coherence protocols. *Formal Methods in System Design*. 2003, 23(3): 257-301
- [42] McMillan K. Modular specification and verification of a cache-coherent interface//Proceedings of the 16th Formal Methods in Computer-Aided Design. Mountain View, USA, 2016: 109-116
- [43] Verbeek F, Yaghini P M, Eghbal A, et al. Advocat: automated deadlock verification for on-chip cache coherence and interconnects//Proceedings of the 19th Design, Automation, Test in Europe Conference, Exhibition. Dresden, Germany, 2016: 1640-1645
- [44] Tan C, Donaldson A F, Wickerson J. Formalising CXL cache coherence//Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. San Jose, USA, 2016: 1640-1645
- [45] Luo L, Zhang Y, He H, et al. Multi-objective optimization regression verification for multi-core cache coherence protocol//Proceedings of the 23rd International Conference on High Performance Computing and Communications. Haikou, China, 2021: 2011-2018
- [46] Xu Xue-Zheng, Yan De-Heng, Wang Lu, et al. Sequential consistency per location theorem proving in RISC-V memory consistency model. *Journal of Software*. 2025, 36(9): 3919-3936 (in Chinese)
(徐学政, 杨德亨, 王璐等. RISC-V 内存一致性模型的同地址顺序一致性定理证明. *软件学报*. 2025, 36(9): 3919-3936)
- [47] Xu Xue-Zheng, Fang Jian, Liang Shao-Jie, et al. Rubyphi: automated model checking for cache coherence protocols in gem5. *Computer Engineering and Science*. 2025, 47(7): 1141-1151 (in Chinese)
(徐学政, 方健, 梁少杰等. Rubyphi: 面向 gem5 的 cache 一致性协议自动化模型检验. *计算机工程与科学*. 2025, 47(7): 1141-1151)

附录 1. 主机端目录 HDC 所涉及的全部瞬时状态(表 2)

表 2 主机端目录控制器 HDC 涉及的全部瞬时状态(37 个)

序号	瞬时状态	语义描述
1	IS_D	初始状态为 I, 目标状态为 S, 等待数据 Data
2	IM_D	初始状态为 I, 目标状态为 M, 等待数据 Data
3	SS_D	初始状态为 S, 目标状态为 S, 等待数据 Data
4	SI_A	初始状态为 S, 目标状态为 I, 等待无效确认消息 InvAck
5	MS_DW	初始状态为 M, 目标状态为 S, 等待数据 Data 和响应消息 Cmp
6	MS_W	初始状态为 M, 目标状态为 S, 等待响应消息 Cmp
7	MI_W	初始状态为 M, 目标状态为 I, 等待响应消息 Cmp
8	II_W	初始状态为 I, 目标状态为 I, 等待响应消息 Cmp
9	SS_W	初始状态为 S, 目标状态为 S, 等待响应消息 Cmp
10	SI_S	初始状态为 S, 目标状态为 I, 等待监听响应消息 RspIHitSE
11	ES_S	初始状态为 E, 目标状态为 S, 等待监听响应消息 RspSHitSE
12	EI_S	初始状态为 E, 目标状态为 I, 等待监听响应消息 RspIHitSE
13	MS_SD	初始状态为 M, 目标状态为 S, 等待监听响应消息 RspSFwdM 和数据 Data
14	MS_D	初始状态为 M, 目标状态为 S, 等待数据 Data
15	MS_S	初始状态为 M, 目标状态为 S, 等待监听响应消息 RspSFwdM
16	MI_SD	初始状态为 M, 目标状态为 I, 等待监听响应消息 RspSFwdM/RspIFwdM 和数据 Data
17	MI_D	初始状态为 M, 目标状态为 I, 等待数据 Data
18	MI_S	初始状态为 M, 目标状态为 I, 等待监听响应消息 RspSFwdM/RspIFwdM
19	IS_SD	初始状态为 I, 目标状态为 S, 等待监听响应消息 RspIHitI/RspSHitSE/RspSFwdM 和数据 Data
20	IS_S	初始状态为 I, 目标状态为 S, 等待监听响应消息 RspSFwdM
21	IM_SD	初始状态为 I, 目标状态为 M, 等待监听响应消息 RspIHitI/RspIHitSE/RspIFwdM 和数据 Data
22	IM_S	初始状态为 I, 目标状态为 M, 等待监听响应消息 RspIFwdM
23	SS_m	初始状态为 S, 目标状态为 S, 等待数据响应消息 MemData 或响应消息 MemAck
24	SM_AS	初始状态为 S, 目标状态为 M, 等待无效确认消息 InvAck 和监听响应消息 RspIHitI/RspIHitSE
25	SM_S	初始状态为 S, 目标状态为 M, 等待监听响应消息 RspIHitI/RspIHitSE
26	SM_A	初始状态为 S, 目标状态为 M, 等待无效确认消息 InvAck
27	MM_m	初始状态为 M, 目标状态为 M, 等待数据响应消息 MemData 或响应消息 MemAck
28	MI_m	初始状态为 M, 目标状态为 I, 等待响应消息 MemAck
29	II_m	初始状态为 I, 目标状态为 I, 等待数据响应消息 MemData 或响应消息 MemAck
30	DSS_m	初始状态为 S, 目标状态为 S, 等待数据响应消息 MemData 或响应消息 MemAck(来自设备端访问流程的瞬时状态)
31	SI_SD	初始状态为 S, 目标状态为 I, 等待响应消息 RspIHitSE 和数据 Data
32	SI_D	初始状态为 S, 目标状态为 I, 等待数据 Data
33	ES_Sm	初始状态为 E, 目标状态为 S, 等待监听响应消息 RspSHitSE 和数据响应消息 MemData
34	ES_m	初始状态为 E, 目标状态为 S, 等待数据响应消息 MemData
35	EI_SD	初始状态为 E, 目标状态为 I, 等待监听响应消息 RspIHitSE 和数据 Data
36	EI_D	初始状态为 E, 目标状态为 I, 等待数据 Data
37	DMS_D	初始状态为 M, 目标状态为 S, 等待数据 Data(来自设备端访问流程的瞬时状态)

附录 2. 主机端目录 HDC 所涉及的全部事务(表 3)

其中, _dev 表示待访问的数据块初始位置存储于设备内存, 而 _host 表示待访问的数据块初始位置存储于主机内存。

表3 主机端目录控制器 HDC 涉及的全部事务(56个)

序号	事务	类型	语义描述
1	GetS_dev	请求	来自 HCC 的读数据请求,目的是请求获取处于 S/E 状态的数据块
2	GetMFrom1S_dev	请求	来自 HCC 的读数据请求,请求者处于 S 状态,且主机端只有一个处于 S 状态的缓存控制器,目的是请求获取处于 M 状态的数据块
3	GetM_dev	请求	来自 HCC 的写数据请求,目的是请求获取处于 M 状态的数据块
4	PutSNotLast_dev	请求	来自 HCC 的 PutS 请求,表明主机端此时有多个处于 S 状态的缓存,目的是将请求者缓存中处于 S 状态的目标数据块置为无效
5	PutSLast_dev	请求	来自 HCC 的 PutS 请求,表明当前请求者是主机端的最后一个处于 S 状态的缓存,目的是将请求者缓存中处于 S 状态的目标数据块置为无效
6	PutEOwner_dev	请求	来自 HCC(为 owner)的 PutE 请求,目的是将请求者缓存中处于 E 状态的数据块置为无效
7	PutENonOwner_dev	请求	来自 HCC(非 owner)的 PutE 请求,目的是将请求者缓存中处于 E 状态的数据块置为无效
8	PutENonOwnerToS_dev	请求	来自 HCC(非 owner)的 PutE 请求,目的是将请求者缓存中处于 E 状态的目标数据块置为无效。注意:当前共享者列表中的共享者个数大于 1,需要将 HDC 状态置为 S
9	PutMOwner_dev	请求	来自 HCC(为 owner)的 PutM 请求,目的是将请求者缓存中处于 M 状态的目标数据块写回设备内存,并将请求者中的目标数据块置为无效
10	PutMNonOwner_dev	请求	来自 HCC(非 owner)的 PutM 请求,目的是将请求者缓存中处于 M 状态的目标数据块置为无效(因为请求者不是 owner,所以无需再将缓存中的数据块写回设备内存)
11	PutMNonOwnerToS_dev	请求	来自 HCC(非 owner)的 PutM 请求,目的是将请求者缓存中处于 M 状态的目标数据块置为无效。注意:当前共享者列表中的共享者个数大于 1,需要将 HDC 状态置为 S
12	Data_dev	响应	来自 HCC(为 owner)的数据响应
13	InvAck_dev	响应	来自 HCC 的无效确认响应,表明该缓存控制器之前收到的无效请求已经被正确处理,但主机端还有其他接收到无效请求的缓存控制器还未返回无效确认响应,需继续等待
14	LastInvAck_dev	响应	来自 HCC 的无效确认响应,表明该缓存控制器之前收到的无效请求已经被正确处理,且主机端所有接收到无效请求的缓存控制器均已返回无效确认响应
15	MemDataCmpE_dev	响应	来自 DDC 的数据响应,提示 HDC 中数据块状态变为 E/M
16	MemDataCmpS_dev	响应	来自 DDC 的数据响应,提示 HDC 中数据块状态变为 S
17	Cmp_dev	响应	来自 DDC 的写回确认响应,表明之前发出的数据写回请求已被正确处理
18	RdAny_dev	请求	来自 DDC 的读请求,目的是获取处于 M/E/S 状态的数据块
19	RdOwnNoData_dev	请求	来自 DDC 的读请求,目的是获取数据块的独占所有权(E/M),而无需获取数据块
20	RdOwn_dev	请求	来自 DDC 的读请求,目的是获取处于 M/E 状态的数据块
21	RspSHitSE_dev	响应	来自 HCC 的响应,表明数据块在该 HCC 中处于 S/E 状态时被命中,并且现已降级为 S 状态
22	RspIHitSE_dev	响应	来自 HCC 的响应,表明数据块在 HCC 中处于 S/E 状态时被命中,并且现已降级为 I 状态
23	RspIHitSELast_dev	响应	来自 HCC 的响应,表明数据块在该 HCC 中处于 S/E 状态时被命中(且这是主机端最后一个处于 S/E 状态的缓存控制器),并且现已降级为 I 状态
24	RspSFwdM_dev	响应	来自 HCC 的响应,表明数据块在该 HCC 中处于 M 状态时被命中,现已降级为 S 状态,并且已将数据块传输至 HDC
25	RspIFwdM_dev	响应	来自 HCC 的响应,表明数据块在该主 HCC 中处于 M 状态时被命中,现在已降级为 I 状态,并且已将数据块传输给 HDC
26	GetS_host	请求	来自 HCC 的读数据请求,目的是请求获取处于 S/E 状态的数据块
27	GetM_host	请求	来自 HCC 的写数据请求,目的是请求获取处于 M 状态的数据块
28	GetMFrom1S_host	请求	来自 HCC 的写数据请求,请求者处于 S 状态,且主机端只有一个处于 S 状态的缓存控制器,目的是请求获取处于 M 状态的数据块
29	PutSNotLast_host	请求	来自 HCC 的 PutS 请求,主机端此时还有不止一个处于 S 状态的缓存,目的是将请求者缓存中处于 S 状态的目标数据块置为无效

(续表)			
序号	事务	类型	语义描述
30	PutSLast_host	请求	来自 HCC 的 PutS 请求,当前请求者是主机端的最后一个处于 S 状态的缓存,目的是将请求者缓存中处于 S 状态的目标数据块置为无效
31	PutEOwner_host	请求	来自 HCC(为 owner)的 PutE 请求,目的是将请求者缓存中处于 E 状态的数据块置为无效
32	PutENonOwner_host	请求	来自 HCC(非 owner)的 PutE 请求,目的是将请求者缓存中处于 E 状态的数据块置为无效
33	PutMOwner_host	请求	来自 HCC(为 owner)的 PutM 请求,目的是将请求者缓存中处于 M 状态的目标数据块写回主机内存,并将请求者中的目标数据块置为无效
34	PutMNonOwner_host	请求	来自 HCC(非 owner)的 PutM 请求,目的是将请求者缓存中处于 M 状态的目标数据块置为无效(因为请求者不是 owner,所以无需再将缓存中的数据块写回主机内存)
35	Data_host	响应	来自 HCC(为 owner)的数据响应
36	InvAck_host	响应	来自 HCC 的无效确认响应,表明该缓存控制器之前收到的无效请求已经被正确处理,且主机端还有其他接收到无效请求的缓存控制器还未返回无效确认响应,需继续等待
37	LastInvAck_host	响应	来自 HCC 的无效确认响应,表明该缓存控制器之前收到的无效请求已经被正确处理,且主机端所有接收到无效请求的缓存控制器均已返回无效确认响应
38	RdAny_host	请求	来自 DDC 的读请求,目的是获取处于 M/E/S 状态的数据块
39	RdOwnNoData_host	请求	来自 DDC 的读请求,目的是获取数据块的独占所有权(E/M),而不需要获取数据块
40	RdOwn_host	请求	来自 DDC 的读请求,目的是获取处于 M/E 状态的数据块
41	RspSHitSE_host	响应	来自 HCC 的响应,表明数据块在该 HCC 中处于 S/E 状态时被命中,并且现已降级为 S 状态
42	RspIHitSE_host	响应	来自 HCC 的响应,表明数据块在该 HCC 中处于 S/E 状态时被命中,并且现已降级为 I 状态
43	RspIHitSElast_host	响应	来自 HCC 的响应,表明数据块在该 HCC 中处于 S/E 状态时被命中(且这是主机端最后一个处于 S/E 状态的缓存控制器),并且现在已降级为 I 状态
44	RspSFwdM_host	响应	来自 HCC 的响应,表明数据块在该 HCC 中处于 M 状态时被命中,现已降级为 S 状态,并且已将数据块传输给 HDC
45	RspIFwdM_host	响应	来自 DCC 的响应,表明数据块在该 HCC 中处于 M 状态时被命中,现已降级为 I 状态,并且已将数据块传输给 HDC
46	MemData_host	响应	来自主机端绑定内存的内存读数据响应
47	MemAck_host	响应	来自主机端绑定内存的内存写入数据响应,表明之前发送到内存的写入内存请求已经被正确执行
48	RspIHitI_host	响应	来自 DDC 的响应,表明数据块在 DDC 中处于 I 状态时被命中,并且现为 I 状态
49	RspSHitSE_host(from DDC)	响应	来自 DDC 的响应,表明数据块在 DDC 中处于 S/E 状态时被命中,并且现已降级为 S 状态
50	RspIHitSE_host(from DDC)	响应	来自 DDC 的响应,表明数据块在 DDC 中处于 S/E 状态时被命中,并且现已降级为 I 状态
51	RspIHitSElast_host(from DDC)	响应	来自 DDC 的响应,表明数据块在 DDC 中处于 S/E 状态时被命中(且设备端只有一个处于 S/E 状态的设备缓存控制器),并且现已降级为 I 状态
52	RspSFwdM_host(from DDC)	响应	来自 DDC 的响应,表明数据块在 DDC 中处于 M 状态时被命中,并且现已降级为 S 状态,且已将数据块传输给 HDC
53	RspIFwdM_host(from DDC)	响应	来自 DDC 的响应,表明数据块在 DDC 中处于 M 状态时被命中,并且现已降级为 I 状态,且已将数据块传输给 HDC
54	CleanEvictNoData_host	请求	来自 DDC 的请求,告知 HDC 设备要将处于 E/S 状态的数据块从设备中丢弃,主机端和设备端没有数据交换
55	DirtyEvict_host	请求	来自 DDC 的请求,将处于 M 状态的数据块从设备中移除,并将修改数据写回主机端内存
56	DataFromDDC_host	响应	来自 DDC 的数据响应

附录3. 设备端目录控制器DDC所涉及的全部瞬时状态(表4)

表4 设备端目录控制器DDC涉及的全部瞬时状态(39个)

序号	瞬时状态	语义描述
1	II_m	初始状态为I,目标状态为I,等待数据响应消息 MemData 或响应消息 MemAck
2	SS_m	初始状态为S,目标状态为S,等待数据响应消息 MemData
3	SI_Sm	初始状态为S,目标状态为I,等待监听响应消息 RspIHitSE 和数据响应消息 MemData
4	SI_S	初始状态为S,目标状态为I,等待监听响应消息 RspIHitSE
5	SI_m	初始状态为S,目标状态为I,等待数据响应消息 MemData
6	ES_Sm	初始状态为E,目标状态为S,等待监听响应消息 RspSHitSE 和数据响应消息 MemData
7	ES_m	初始状态为E,目标状态为S,等待数据响应消息 MemData
8	ES_S	初始状态为E,目标状态为S,等待监听响应消息 RspSHitSE
9	EI_Sm	初始状态为E,目标状态为I,等待监听响应消息 RspIHitSE 和数据响应消息 MemData
10	EI_S	初始状态为E,目标状态为I,等待监听响应消息 RspIHitSE
11	EI_m	初始状态为E,目标状态为I,等待数据响应消息 MemData
12	MS_Sm	初始状态为M,目标状态为S,等待监听响应消息 RspSFwdM/RspIFwdM 和响应消息 MemAck
13	MS_m	初始状态为M,目标状态为S,等待响应消息 MemAck
14	MI_S	初始状态为M,目标状态为I,等待监听响应消息 RspIFwdM
15	IS_FD	初始状态为I,目标状态为S,等待转发消息 MemRdFwdI/MemRdFwdS 和数据 Data
16	IS_FDm	初始状态为I,目标状态为S,等待转发消息 MemRdFwdI/MemRdFwdS、响应消息 MemData/MemAck 和数据 Data
17	IS_D	初始状态为I,目标状态为S,等待数据 Data
18	IE_D	初始状态为I,目标状态为E,等待数据 Data
19	IM_FD	初始状态为I,目标状态为M,等待转发消息 MemRdFwdI 和数据 Data
20	IM_FDm	初始状态为I,目标状态为M,等待转发消息 MemRdFwdI、响应消息 MemAck 和数据 Data
21	IM_D	初始状态为I,目标状态为M,等待数据 Data
22	IM_Dm	初始状态为I,目标状态为M,等待数据 Data 和响应消息 MemAck
23	MM_m	初始状态为M,目标状态为M,等待数据响应消息 MemData
24	SM_F	初始状态为S,目标状态为M,等待转发消息 MemRdFwdI
25	SM_Fm	初始状态为S,目标状态为M,等待转发消息 MemRdFwdI 和数据响应消息 MemData
26	MI_m	初始状态为M,目标状态为I,等待响应消息 MemAck
27	MS_SD	初始状态为M,目标状态为S,等待监听响应消息 RspIFwdM 和数据 Data
28	MS_D	初始状态为M,目标状态为S,等待数据 Data
29	MS_S	初始状态为M,目标状态为S,等待监听响应消息 RspIFwdM
30	MI_SD	初始状态为M,目标状态为I,等待监听响应消息 RspIFwd 和数据 Data
31	MI_D	初始状态为M,目标状态为I,等待数据 Data
32	IS_GD	初始状态为I,目标状态为S,等待响应消息 GO-S/GO-E 和数据 Data
33	IS_G	初始状态为I,目标状态为S,等待响应消息 GO-S/GO-E
34	IM_GD	初始状态为I,目标状态为M,等待响应消息 GO-E/GO-M 和数据 Data
35	IM_G	初始状态为I,目标状态为M,等待响应消息 GO-E/GO-M
36	SM_G	初始状态为S,目标状态为M,等待响应消息 GO-M
37	EI_G	初始状态为E,目标状态为I,等待响应消息 GO-I
38	SI_G	初始状态为S,目标状态为I,等待响应消息 GO-I
39	MI_G	初始状态为M,目标状态为I,等待响应消息 GO-WritePull

附录4. 设备端目录控制器DDC所涉及的全部事务(表5)

表5 设备端目录控制器DDC涉及的全部事务(37个)			
序号	事务	类型	语义描述
1	MemRdSnpData_dev	请求	来自HDC的读数据请求,目的是从设备端获取数据,并将设备端处于E/M状态的数据块状态置为S
2	MemRdSnpInv_dev	请求	来自HDC的读数据和无效请求,目的是从设备端获取数据,并将设备端的处于M/E/S状态的数据块状态置为无效I
3	MemWrSnp_dev	请求	来自HDC的写数据请求,目的是将数据写回设备端绑定内存
4	RspSHitSE_dev	响应	来自DCC的响应,表明数据块在设备缓存中处于S/E状态时被命中,且现已降级为S状态
5	RspIHitSE_dev	响应	来自DCC的响应,表明数据块在设备缓存中处于S/E状态时被命中,且现已降级为I状态
6	RspSFwdM_dev	响应	来自DCC的响应,表明数据块在设备缓存中处于M状态时被命中,现已降级为S状态,并且已将数据块传输给DDC
7	RspIFwdM_dev	响应	来自DCC的响应,表明数据块在设备缓存中处于M状态时被命中,现已降级为I状态,并且已将数据块传输给DDC
8	MemData_dev	响应	来自设备内存控制器的数据响应
9	MemAck_dev	响应	来自设备内存控制器的写入确认响应,表明内存控制器已经将要写入的数据成功写入内存
10	GetS_dev	请求	来自DCC的读数据请求,目的是请求获取处于S/E状态的数据块
11	GetM_dev	请求	来自DCC的写数据请求,目的是请求获取处于M状态的数据块
12	PutSLast_dev	请求	来自DCC的PutS请求,当前请求者是设备端的最后一个存有S状态的缓存,目的是将请求者缓存中处于S状态的目标数据块置为无效
13	PutEOwner_dev	请求	来自DCC(为owner)的PutE请求,目的是将请求者缓存中处于E状态的目标数据块置为无效
14	PutENonOwner_dev	请求	来自DCC(非owner)的PutE请求,目的是将请求者缓存中处于E状态的目标数据块置为无效
15	PutMOwner_dev	请求	来自DCC(为owner)的PutM请求,目的是将请求者中处于M状态的目标数据块写回设备内存,并将请求者中缓存的目标数据块置为无效
16	PutMNonOwner_dev	请求	来自DCC(非owner)的PutM请求,目的是将请求者中缓存的处于M状态的目标数据块置为无效(因为请求者不是owner,所以无需再将缓存中的数据块写回设备内存)
17	MemRdFwdI_dev	转发	来自HDC的转发消息,表明当前数据块在主机端状态为I,数据块可以直接从设备绑定内存转发到设备缓存,而无需向主机发送任何完成通知
18	MemRdFwdS_dev	转发	来自HDC的转发消息,表明当前数据块在主机端状态为S,数据块可以直接从设备绑定内存转发到设备缓存,而无需向主机发送任何完成通知
19	RspSHitSE_host	响应	来自DCC的响应,表明数据块在设备缓存中处于S/E状态时被命中,且现已降级为S状态
20	RspIHitSE_host	响应	来自DCC的响应,表明数据块在设备缓存中处于S/E状态时被命中,且现已降级为I状态
21	RspSFwdM_host	响应	来自DCC的响应,表明数据块在设备缓存中处于M状态时被命中,现已降级为S状态,并且已将数据块传输给DDC
22	RspIFwdM_host	响应	来自DCC的响应,表明数据块在设备缓存中处于M状态时被命中,现已降级为I状态,并且已将数据块传输给DDC
23	GetS_host	请求	来自DCC的读数据请求,目的是请求获取处于S/E状态的数据块
24	GetM_host	请求	来自DCC的写数据请求,目的是请求获取处于M状态的数据块
25	SnpData_host	请求	来自HDC的监听数据请求,目的是将设备端处于E/M状态的数据块降级为S,如果设备持有修改数据,则必须将其返回给主机
26	SnpInv_host	请求	来自HDC的监听无效请求,目的是将设备端的处于M/E/S状态的数据块状态降级为无效I
27	Data_host	响应	来自DCC的数据响应
28	PutS_host	请求	来自DCC的PutS请求,目的是将请求者中处于S状态的数据块置为无效
29	PutE_host	请求	来自DCC的PutE请求,目的是将请求者中处于E状态的数据块置为无效
30	PutM_host	请求	来自DCC的PutM请求,目的是将请求者中处于M状态的数据写回主机内存,并将请求者中的目标数据块状态置为无效
31	GoI_host	响应	来自HDC的GO响应,指示了数据块在DDC中应置于I状态
32	GoS_host	响应	来自HDC的GO响应,指示了数据块在DDC中应置于S状态
33	GoE_host	响应	来自HDC的GO响应,指示了数据块在DDC中应置于E状态
34	GoM_host	响应	来自HDC的GO响应,指示了数据块在DDC中应置于M状态
35	GoWritePull_host	响应	来自HDC的GO与WritePull组合响应,表明数据块应在DDC中应置于I状态,同时返回修改数据给HDC
36	DataFromHDC_host	响应	来自HDC的数据响应
37	ExclusiveDataFromHDC_host	响应	来自HDC的独占数据响应

附录5. 引理1的证明

设 $q_0 = \text{head}(Q_{\text{Resp}}(s))$ 为响应队列的队首消息。由迁移规则, 对任意消息 $q \in \sum_{\text{Resp}}$, 存在规则 R_q 使得若 $q = \text{head}(Q_{\text{Resp}}(s))$ 且前提 $\text{pre}_q(s)$ 成立, 则 $\text{enabled}(\sigma_q, s)$ 对相应的 $\sigma_q \in \sum_{\text{Resp}}$ 成立。由网络优先级设计, 响应消息的产生不依赖于其他未完成消息, 故 $\text{pre}_{q_0}(s)$ 必然成立。因此 $\left(\bigvee_{\sigma \in \sum_{\text{Resp}}} \text{enabled}(\sigma, s) \right)$ 成立。即响应队列非空必然意味着至少一个响应动作是使能的(可发生的)。

附录6. 引理2的证明

设 $q_0 = \text{head}(Q_{\text{Fwd}}(s))$ 为转发队列的队首消息。存在规则 R_q 与动作 $\sigma_q \in \sum_{\text{Fwd}}$ 关联。若前提条件 $\text{pre}_q(s)$ 成立, 则 $\text{enabled}(\sigma_q, s)$ 成立。若 $\text{pre}_q(s)$ 不成立, 则系统处理等待条件状态。由协议设计, 转发消息处理条件最终会满足, 且此等待过程不依赖响应列。因此存在从状态 s 出发的有限迁移序列 $s \xrightarrow{\sigma_1} \dots \xrightarrow{\sigma_k} s_k$ 使得 $\text{pre}_q(s_k)$ 成立, 且该序列中所有动作均使能于各自状态。特别地, 动作 σ_1 在状态 s 下使能, 且 $\sigma_1 \in \sum_{\text{Fwd}} \cup \sum_{\text{Req}}$ 。若 $\sigma_1 \in \sum_{\text{Fwd}}$, 则结论得证。若 $\sigma_1 \in \sum_{\text{Req}}$, 则由非空闲状态下存在使能请求动作, 故总有 $\left(\bigvee_{\sigma \in \sum_{\text{Fwd}}} \text{enabled}(\sigma, s) \right)$ 成立。即在任何可达状态 s 下, 如果响应队列为空但转发队列非空, 则至少存在一个转发类动作是使能的。

附录7. 引理3的证明

定义系统空闲状态 $\text{Idle}(s) \equiv (Q_{\text{Resp}}(s) = \emptyset \wedge Q_{\text{Fwd}}(s) = \emptyset \wedge \forall i, a: C_{\text{HCC}}^i(a) \neq I \wedge \forall j, a: C_{\text{DCC}}^j(a) \neq I)$ 。若系统处于非空闲状态 $\neg \text{Idle}(s)$, 则存在某个缓存控制器(HCC或DCC)对地址 a 状态为 I (未命中)。由迁移规则, 该控制器可立即发起请求动作(如 GetS 、 GetM), 故 $\left(\bigvee_{\sigma \in \sum_{\text{Req}}} \text{enabled}(\sigma, s) \right)$ 成立。即在任何可达状态 s 下, 如果响应队列和转发队列均为空, 则至少存在一个请求类动作是使能的。

附录8. 定理1的证明

对任意从初始状态 s_0 出发的可达状态 s , 现考虑以下三种情况:

- (1): 如果响应队列不为空, 即 $Q_{\text{Resp}}(s) \neq \emptyset$, 再根据引理1可知 $\bigvee_{\sigma \in \sum_{\text{Resp}}} \text{enabled}(\sigma, s)$ 成立, 因此有 $\bigvee_{\sigma \in \Sigma} \text{enabled}(\sigma, s)$ 成立;
- (2): 如果响应队列为空并且转发队列不为空, 即 $Q_{\text{Resp}}(s) = \emptyset \wedge Q_{\text{Fwd}}(s) \neq \emptyset$, 再根据引理2可知, $\bigvee_{\sigma \in \sum_{\text{Fwd}}} \text{enabled}(\sigma, s)$ 成立, 因此有 $\bigvee_{\sigma \in \Sigma} \text{enabled}(\sigma, s)$ 成立;
- (3): 如果响应队列为空且转发队列也为空, 则根据引理3可知 $\bigvee_{\sigma \in \sum_{\text{Req}}} \text{enabled}(\sigma, s)$ 成立, 因此有 $\bigvee_{\sigma \in \Sigma} \text{enabled}(\sigma, s)$ 成立。

综上所述, 由计算树逻辑语义, $\mathcal{M}, s_0 \models \text{AG} \left(\bigvee_{\sigma \in \Sigma} \text{enabled}(\sigma, s) \right)$ 在所有可达状态 s 下成立, 即协议不存在死锁。

附录9. 定理2的证明

(1) 归纳基础: 初始状态为 s_0 , 对所有地址 a 及处理器 i 与设备 j , 有以下表达式成立:

$$C_{\text{HDC}}(a).stab = I, C_{\text{HCC}}^i(a) = I, C_{\text{DDC}}(a).stab = I, \\ C_{\text{DCC}}^j(a) = I.$$

将上述条件代入聚合函数后可得 $\text{Agg}_{\text{host}}(a) = I$ 并且 $\text{Agg}_{\text{dev}}(a) = I$ 。因此, $\text{DirC}_{\text{host}}(a)$ 与 $\text{DirC}_{\text{dev}}(a)$ 在 s_0 处成立。

(2) 归纳假设: 假设对于某个可达状态 s , 对所有地址 a , 目录一致性成立。即:

$$\forall a \in \mathcal{A}: C_{\text{HDC}}(a).stab = \\ \text{Agg}_{\text{host}}(a) \wedge C_{\text{DDC}}(a).stab = \text{Agg}_{\text{dev}}(a)^\circ$$

(3) 归纳步骤: 考虑任意一个从状态 s 出发的迁移 $s \xrightarrow{\sigma} s'$, 只需要证明在迁移后的状态 s' , 目录一致性对所有地址 a 仍然成立。此时, 需要对动作 σ 分类讨论。

为简化证明, 下面以 $\sigma = \text{GetM}(a, i)$ 为例进行证明。

由迁移规则在 s 状态下, 有条件 $C_{\text{HCC}}^i(a) = I$ 和 $C_{\text{HDC}}(a).stab = I$ 成立。执行动作 $\sigma = \text{GetM}(a, i)$ 后, 状态 s' 满足条件: $C_{\text{HCC}}^i(a) = M, \forall k \neq i: C_{\text{HCC}}^k(a) = I, \forall j: C_{\text{DCC}}^j(a) = I, C_{\text{HDC}}'(a).stab = M(\text{owner} = i)$, 并且 $C_{\text{DDC}}'(a).stab = I$ 。

由聚合函数计算可得, $\text{Agg}_{\text{host}}(a, s') = M$, 这是因为 $\exists i: C_{\text{HCC}}^i(a) = M$ 。另外, $\text{Agg}_{\text{dev}}(a, s') = I$, 因为 $\forall j: C_{\text{DCC}}^j(a) = I$ 并且 $C_{\text{HDC}}'(a).stab = M$ (远程 M 状态不被设备聚合函数视为本地 M)。

(4) 结论: 基于上述推理, $C_{\text{HDC}}(a) = \text{Agg}_{\text{host}}(a, s')$ 满足 $\text{DirC}_{\text{host}}(a)$, 同理 $C_{\text{DDC}}(a) = \text{Agg}_{\text{dev}}(a, s')$ 也满足 $\text{DirC}_{\text{dev}}(a)$ 。其他动作采用类似方法可证。由归纳法可得 $\forall s \rightarrow \text{DirC}(s)$, 即定理2成立。

附录10. 定理3的证明

(1) 写者唯一性。假设存在地址 a 使得 $\text{WCount}(a, s) > 1$, 则有以下两种情况:

- ①: $\exists i \neq k: C_{\text{HCC}}^i(a) = M \wedge C_{\text{HCC}}^k(a) = M$;
- ②: $\exists i: C_{\text{HCC}}^i(a) = M \wedge C_{\text{DDC}}(a).stab = M$ 。

由定理2的目录一致性 $\text{DirC}(s)$ 可知, $C_{\text{HDC}}(a).stab = \text{Agg}_{\text{host}}(a, s)$ 。如果 Case 1 成立, 则 $\text{Agg}_{\text{host}}(a, s) = M$, 但聚合函数定义要求主机端至多存在一个 M 状态, 故矛盾。如果 Case 2 成立, 则 $C_{\text{HDC}}(a).stab = M$, 并且拥有者为某个 CPU; 同时 $C_{\text{DDC}}(a).stab = M$ 。但聚合函数定义中, 当 $C_{\text{DDC}}(a).stab = M$ 且拥有者为非 DDC 时, $\text{Agg}_{\text{host}}(a, s)$ 应为状态 I 或 S , 与 $C_{\text{HDC}}(a).stab = M$ 矛盾。故原假设不成立, 即 $\text{WCount}(a, s) \leq 1$ 。

(2) 读写互斥。如果 $\text{WCount}(a, s) = 1$, 则 $C_{\text{HDC}}(a).stab = M$ 或 $C_{\text{DDC}}(a).stab = M$ 。由聚合函数可知, 此时不可能有缓存行状态为 S (否则聚合结果为 S), 故 $\text{RCount}(a, s) = 0$ 。如果

$RCount(a, s) \geq 1$, 则 $Agg_{host}(a, s) = S$ 或 $Agg_{dev}(a, s) = S$, 故 $C_{HDC}(a).stab \neq M$

且 $C_{DDC}(a).stab \neq M$, 即 $WCount(a, s) = 0$ 。

综上所述, 由 $DirC(s) \rightarrow \forall a: SWMR(a, s)$ 。进一步地, $AG(\forall a \in \mathcal{A}: SWMR(a, s))$ 成立。

附录 11. 定理 4 的证明

(1) 不变式定义如下:

$$DataInv(s) \equiv$$

$$\left\{ \begin{array}{l} \forall a \in \mathcal{A}: (\exists! w: IsWriter(w, a, s) \wedge data(w, a, s) = v_{latest}), \\ \vee (\neg \exists w: IsWriter(w, a, s) \wedge (Mem(a) = v_{latest})), \\ \wedge (\forall r: IsReader(r, a, s) \rightarrow data(r, a, s) = v_{latest}), \\ \wedge (\forall q \in \sum_{Read_{req}}: data(a, s) = v_{latest}) \end{array} \right.$$

其中, $IsWriter(w, a, s)$ 表示组件 w (缓存或目录) 是地址 a 的唯一-写者 (状态为 M)。 $IsReader(r, a, s)$ 表示组件 r 是地址 a 的读者。 $\sum_{Read_{req}}$ 表示读请求队列。

上述不变式断言: 一是最新值要么被唯一的写者持有, 要么在内存中; 二是所有读者 (S 状态) 持有的值都是该最新值; 三是任何读操作最终获取到的值都是该最新值。

(2) 归纳基础: 在初始状态 s_0 时无写者, 所有读者 (无) 数据与内存保持一致, $DataInv(s_0)$ 成立。

(3) 归纳假设: 在可达状态 s , 对任意地址 a , $DataInv(s)$ 成立。

(4) 归纳步骤: 考虑迁移规则 $s \xrightarrow{\sigma} s'$, 并对动作 σ 进行分类讨论:

①: 写授权动作 (如 GetM 后接数据写入)。由 SWMR 性质, 授权前无效化所有其他副本。新写者数据来自当前最新值持有者 (由 $DataInv(s)$ 保证)。写入后, 新写者持有新最新值, 无读者, 内存可能过时, $DataInv(s')$ 成立。

②: 读动作 (如 GetS)。若命中, 直接返回缓存数据, 由 $DataInv(s)$ 该数据为最新值; 若未命中, 协议从最新值持有者获取数据并返回, 同时可能新增读者, $DataInv(s')$ 成立。

③: 替换/写回动作 (如 PutM)。写者数据写回内存, 内存更新为最新值, 写者消失, $DataInv(s')$ 成立。

④: 数据传递动作。按协议规则从正确源传递数据到目标, 保持最新值传播。

(5) 结论: 通过对所有动作类型的分析, 可以得出结论: 如果 $DataInv(s)$ 在迁移前成立, 那么在迁移后也成立。因此, 由归纳法, $DataInv(s)$ 在所有可达状态成立。因此, $DataInv(s) \rightarrow \forall a: DVI(a, s)$ 。故一致性协议满足数据值不变性。



LI Ren-Gang, Ph. D., professor-level senior engineer. His research interests include computer architecture design and artificial intelligence.

YAN Rui-Dong, Ph. D., senior engineer. His research interests include computer architecture design and the optimization algorithms of the distributed training system.

GUO Zhen-Hua, Ph. D., professor-level senior engineer. His main research interest includes computer architecture design.

ZHAI Ji-Dong, Ph. D., professor, Ph. D. supervisor. His research interests include compilation optimization, high performance computing, and artificial intelligence.

WANG Li, M. S., senior engineer. Her main research interest includes computer architecture design.

CAO Fang, M. S., senior engineer. Her main research interest includes computer architecture design.

GAO Kai, M. S. His main research interest includes computer architecture design.

ZHANG Hong-Wei, Ph. D. His main research interest includes computer architecture design.

YANG Guang-Wen, Ph. D., professor, PhD supervisor. His main research interest includes high performance computing.

Background

In recent years, deep learning applications, particularly in natural language processing and computer vision, have achieved remarkable progress. The dramatic expansion of data scales and model complexity has led to an exponential growth in computational demands. Against this backdrop, heterogeneous computing systems have gradually emerged as the mainstream computing architecture, owing to their efficiency in collaborative

computing and favorable scalability. However, the integration of diverse heterogeneous computing units and memory components in such systems has given rise to the “memory wall” effect. To address this challenge, researchers from both academia and industry have conducted extensive studies on the design and verification of cache coherence protocols, achieving incremental breakthroughs. Existing hierarchical protocol conversion and

translation frameworks employ logically complex proxy caches or translators to construct various novel composite coherence protocols. Nevertheless, these protocols involve frequent operations such as message parsing, decision-making, and transformation, thereby increasing the complexity of protocol design and verification while introducing additional performance overhead. Their scalability and compatibility are considerably constrained.

Specifically, heterogeneous computing systems face the following three technical challenges:

(1) Challenges in cache coherence protocol design. Heterogeneous computing units are equipped with their own specific cache coherence protocols, and their divergent communication paradigms exacerbate the complexity of protocol design. For instance, the classic MESI protocol employs an invalidation mechanism initiated by readers and writers to enforce the Single-Writer-Multiple-Reader (SWMR) invariant. This mechanism is suitable for CPUs but not for GPUs.

(2) Challenges in directory controller design. As system scale continues to expand, data interaction among heterogeneous computing units becomes increasingly frequent, leading to a growth in shared cache information. Consequently, employing a directory structure to maintain coherence has become an inevitable choice. However, divergent protocols result in differing internal directory structures and stored contents across computing units, which in turn leads to variations in the processing flows of

coherence requests and responses.

(3) Challenges in cache coherence protocol verification. Factors such as the increasing scale of systems, the diversity of coherence protocols, and the complexity of interconnection networks pose significant challenges to verifying the correctness and completeness of these protocols.

To address the aforementioned challenges, this paper proposes a cache coherence maintenance method for heterogeneous computing systems based on a unified state representation, which comprises two key components: protocol design and formal verification. The former supports efficient memory access among heterogeneous computing units: reducing latency and bandwidth overhead; the latter employs a symbolic theorem proving approach to theoretically verify critical properties such as deadlock-freedom of the system state machine and the SWMR invariant. To evaluate the effectiveness of the proposed protocol, we implemented it in the gem5 simulator and tested its transaction coverage and latency performance under various memory access paths. Furthermore, comparative experiments were conducted on multiple SPLASH-2 benchmark datasets. Experimental results demonstrate that the proposed protocol can effectively support data access among heterogeneous computing units, with its runtime performance differing from that of the classic MESI protocol by only 2% to 4%.

This work is supported by The National Key Research and Development Program of China No. 2024YFE0210800.