PR #34519 完整报告

sgl-project/sglang

fix(hicache): limit load-back pending to write-back

合并时间：2026-08-17 14:23

原文链接：http://prhub.com.cn/sgl-project/sglang/pull/34519

执行摘要

- 一句话：load-back 挂起标记限定为 write-back 模式，修复 write-through 并发伪断言
- 推荐动作：值得精读（约 120 行改动）。这是一个典型的「通过收窄保护触发条件消除伪冲突，而不是引入更多并发控制」的案例：先追溯保护标记的原始目的（防止 write-back 重复回收踩到未完成 DMA），再论证 write-through 为何不需要该保护（回收入口不存在 + Host 锁仍持有），最后保留 ACK 时必要的重复跟踪。对从事 HiCache、radix cache 并发控制或「锁 / 标记的作用域应与风险域对齐」这类设计权衡的工程师有参考价值。关注 finish_load_back() 的分策略 ACK 逻辑与两个边界测试的断言写法。

功能与动机

load_back_pending_id 最初与 write-back 重复 Host 副本回收一同引入：在 write-back 模式下节点可能同时在 GPU 与 Host 持有 Full KV，Host 内存紧张时 _reclaim_full_host_duplicates() 可能释放冗余 Host 副本，而 H→D load-back 仍在读取这些槽位，因此需要把源节点钉住直到 ACK。但原实现对所有写策略无条件安装该标记，导致 write-through 模式下两条不同 anchor 的 load-back 在相同源节点重叠时触发 AssertionError: node N pinned by load-back A, new anchor B。PR body 明确指出该断言在 write-through 下是多余的：Full Host 重复回收只在 self.is_write_back 为 true 时进入，且 load_back() 获取的 Host 锁在 DMA ACK 前保持持有。受影响部署使用了 --hicache-write-policy write_through。

实现拆解

本 PR 的核心是把 load_back_pending_id 的职责边界从「所有 write policy」收窄到「仅 write-back」，并通过调整 ACK 路径保证两种模式的重复跟踪语义不退化。

1. 限定 commit_load_back() 的钉住范围（python/sglang/srt/mem_cache/unified_cache/unified_tree_core.py）：将原先无条件执行的「遍历 kv_xfer 与 comp_xfers 的 nodes_to_load，断言并设置 pinned.load_back_pending_id」逻辑整体包进 if self.is_write_back: 分支。这样 write-through 模式下不同 anchor 的 load-back 可以在同一源节点上自由重叠，不再触发虚假断言；write-back 模式下原有「同一节点同一时刻仅允许一个 anchor 钉住」的保守语义原样保留。
2. 重构 finish_load_back() 的 ACK 收尾：原先只在 node.load_back_pending_id == anchor_node_id 时清除标记并调用 _update_duplicate_tracking(node)；现在按策略分流——write-back 分支只在匹配 anchor 时清除标记（不匹配直接跳过该节点），

write-through 分支不处理标记但无条件对路径上每个节点调用 _update_duplicate_tracking(node)。这一步是关键正确性修复：存储预取填充的节点可能从 L3 获得 Host KV，并在 load-back 中首次形成 GPU+Host 重复，且从未经过本地 write-through 备份 ACK，若不刷新会触发空闲不变量检查器的 Duplicate missing 报错。

3. 新增 CPU 单测覆盖两个边界（test/registered/unit/mem_cache/test_unified_radix_cache_unittest.py）：新增 TestUnifiedTreeCoreLoadBackPending 测试类，用 mock.Mock 构造 root -> shared -> (anchor_a, anchor_b) 树形 core，并把 _is_settled_full_host_duplicate、_update_duplicate_tracking 绑定到真实实现。两个测试分别验证：write-through 下双 anchor 重叠 load-back 不安装标记且 ACK 后重复被跟踪；write-back 下 pending 标记在 ACK 前阻止回收、二次不同 anchor 仍断言、ACK 后恢复可回收。测试结果 52 passed，并附有 openai/gpt-oss-20b 的 H100 端到端验证（20/20 L2 load-back 成功、无 Duplicate missing、HTTP 200）。

关键文件：

- python/sglang/srt/mem_cache/unified_cache/unified_tree_core.py（模块 缓存核心；类别 source；类型 core-logic；符号 commit_load_back, finish_load_back）：核心源码改动文件。commit_load_back() 将 load_back_pending_id 的钉住逻辑限制到 write-back 模式，finish_load_back() 按写策略分流 ACK 处理并保证 write-through 仍刷新重复跟踪。
- test/registered/unit/mem_cache/test_unified_radix_cache_unittest.py（模块 单元测试；类别 test；类型 test-coverage；符号 TestUnifiedTreeCoreLoadBackPending, _build_core, _commit_load_back, test_write_through_different_anchors_track_duplicate_without_pending）：新增 TestUnifiedTreeCoreLoadBackPending 测试类，用 mock core 模拟两种写策略下 load-back 的重叠与回收边界，直接锁定本 PR 的修复语义。

关键符号：commit_load_back, finish_load_back

关键源码片段

python/sglang/srt/mem_cache/unified_cache/unified_tree_core.py

核心源码改动文件。commit_load_back() 将 load_back_pending_id 的钉住逻辑限制到 write-back 模式，finish_load_back() 按写策略分流 ACK 处理并保证 write-through 仍刷新重复跟踪。

```
def commit_load_back(
    self, node_id, device_indices, kv_xfer, comp_xfers
):
    """提交一次成功的 H->D load-back；SWA 的 full->swa 映射重建延迟到编排层。"""
    node = self.node_by_id(node_id)
    cache_actions = []
    if self.is_write_back:
        # write-back 下 Host 重复副本可能在 DMA 未完成时被回收，
        # 因此必须把每个源节点钉住直到 ACK。write-through 不需要
        # 该保护：其 Full Host 重复回收只在 write-back 模式进入，
        # 且 load_back() 持有的 Host 锁在 ACK 前不会释放。
        for xfers in ([kv_xfer], *comp_xfers.values()):
            for xfer in xfers:
```

```
                for nid in xfer.nodes_to_load or ():
                    pinned = self.node_by_id(nid)
                    # 同一时刻每个节点只允许一个 load-back 挂起；
                    # 同一 anchor 可重复钉住（节点可能同时出现在
                    # Full 与 aux 传输列表中）。
                    assert pinned.load_back_pending_id in (None, node_id), (
                        f"node {nid} pinned by load-back "
                        f"{pinned.load_back_pending_id}, new anchor {node_id}"
                    )
                    pinned.load_back_pending_id = node_id
        kv_xfer.device_indices = device_indices
        self.components_by_type[BASE_COMPONENT_TYPE].commit_hicache_transfer(
            node, CacheTransferPhase.LOAD_BACK, [kv_xfer], cache_actions
        )
        # ... 其余组件 transfer 与 store event 记录省略 ...
        self._update_evictable_leaf_sets(node)
        return cache_actions


    def finish_load_back(self, anchor_node_id):
        """ACK 时沿 anchor 根路径收尾。

        write-back 清除源节点钉住标记；write-through 不安装标记，
        但两者都必须刷新重复跟踪：存储预取节点可能从 L3 获得 Host KV，
        在此次 load-back 中才形成首个 GPU+Host 重复，且从未经过本地
        write-through 备份 ACK，跳过会触发 "Duplicate missing" 错误。
        """
        node = self.node_by_id(anchor_node_id)
        while node is not None and node is not self.root_node:
            if self.is_write_back:
                if node.load_back_pending_id != anchor_node_id:
                    node = node.parent
                    continue
                node.load_back_pending_id = None
            self._update_duplicate_tracking(node)
            node = node.parent
```

test/registered/unit/mem_cache/test_unified_radix_cache_unittest.py

新增 TestUnifiedTreeCoreLoadBackPending 测试类，用 mock core 模拟两种写策略下 load-back 的重叠与回收边界，直接锁定本 PR 的修复语义。

```
    class TestUnifiedTreeCoreLoadBackPending(CustomTestCase):
        """覆盖 load-back pending 标记在两种写策略下的边界行为（CPU 单测）。"""

        def _build_core(self, *, is_write_back: bool):
            # 用 mock core 代替完整 UnifiedRadixCache，只保留被测方法依赖的成员；
            # 树结构：root -> shared，shared 下挂 anchor_a / anchor_b 两个 anchor，
            # 两个 anchor 可对同一 shared 节点发起 load-back，从而测出重叠场景。
            component_types = (ComponentType.FULL,)
```

```
        root = UnifiedTreeNode(component_types)
        shared = UnifiedTreeNode(component_types)
        anchor_a = UnifiedTreeNode(component_types)
        anchor_b = UnifiedTreeNode(component_types)
        shared.parent = root
        anchor_a.parent = shared
        anchor_b.parent = shared

        nodes = {node.id: node for node in (root, shared, anchor_a, anchor_b)}
        core = mock.Mock()
        core.is_write_back = is_write_back
        core.root_node = root
        core.node_by_id.side_effect = nodes.__getitem__
        # 关键：把真实实现绑定到 mock 上，保证被测的是生产逻辑而非桩。
        core._is_settled_full_host_duplicate.side_effect = (
            lambda node: UnifiedTreeCore._is_settled_full_host_duplicate(core, node)
        )
        core._update_duplicate_tracking.side_effect = (
            lambda node: UnifiedTreeCore._update_duplicate_tracking(core, node)
        )
        return core, shared, anchor_a, anchor_b

    def test_write_through_different_anchors_track_duplicate_without_pending(self):
        # write-through：两个不同 anchor 可重叠 load-back 同一源节点，
        # 不安装 pending 标记、不触发断言；ACK 后重复被正确跟踪。
        core, shared, anchor_a, anchor_b = self._build_core(is_write_back=False)
        self._commit_load_back(core, anchor_a, shared)
        self._commit_load_back(core, anchor_b, shared) # 旧行为在此会断言失败
        UnifiedTreeCore.finish_load_back(core, anchor_a.id)
        self.assertIsNone(shared.load_back_pending_id)
        self.assertIn(shared.id, core.full_host_duplicates)

    def test_write_back_pending_blocks_reclaim_until_ack(self):
        # write-back：pending 标记在 ACK 前阻止 Host 重复回收，
        # 不同 anchor 的二次 load-back 仍被拒绝；ACK 后恢复可回收。
        core, shared, anchor_a, anchor_b = self._build_core(is_write_back=True)
        self._commit_load_back(core, anchor_a, shared)
        self.assertEqual(shared.load_back_pending_id, anchor_a.id)
        self.assertFalse(UnifiedTreeCore._can_reclaim_full_host_duplicate(core, shared))
        with self.assertRaisesRegex(AssertionError, "new anchor"):
            self._commit_load_back(core, anchor_b, shared)
        UnifiedTreeCore.finish_load_back(core, anchor_a.id)
        self.assertIsNone(shared.load_back_pending_id)
        self.assertTrue(UnifiedTreeCore._can_reclaim_full_host_duplicate(core, shared))
```

评论区精华

核心讨论围绕「收窄保护范围是否安全」展开：

- hzh0425 在 finish_load_back() 的改动处直接提问：> why need this change? PR body 给出了明确答复：write-through 模式下存储预取节点可能在没有经过本地 write-through 备份 ACK 的情况下形成首个 GPU+Host 重复，因此 ACK 时必须刷新重复跟踪，否则空闲不变量检查器报 Duplicate missing。
- xiezhq-hermann 在 Issue 评论中询问：> can you help check whether this PR fixes this problem as well? https://github.com/sgl-project/sglang/pull/34046 即本 PR 是否也能顺带修复 PR #34046 的问题。
- hzh0425 在合并前明确划定了范围：> Let's limit this check to write_back mode first, without affecting write_through mode write_back mode needs to be fixed by @xiezhq-hermann later 即先消除 write-through 的误伤，write-back 模式更深层的并发语义问题留给后续 PR 处理。两人均 APPROVED。此外 hzh0425 还用截图说明部分失败测试与本 PR 无关并触发了 /rerun-test 确认为绿色。
- finish_load_back 为何要在 write-through 分支也刷新重复跟踪 (question): PR body 解释：存储预取填充的节点可能从 L3 获得 Host KV，并在 load-back 中首次形成 GPU+Host 重复，且从未经过本地 write-through 备份 ACK；若不刷新会导致空闲不变量检查器报 Duplicate missing。
- 本 PR 是否覆盖 write-back 模式遗留问题（PR #34046） (question): 范围明确：write-through 的误伤由本 PR 修复；write-back 的并发语义问题不在本 PR 内，后续单独跟进。

风险与影响

- 风险：
  1. write-back 并发语义遗留：PR 明确保留了 write-back 下「不同 anchor 重叠 load-back 仍断言」的保守行为，review 中确认 PR #34046 涉及的 write-back 问题不在本 PR 范围内，后续由 xiezhq-hermann 跟进，当前未解决。
  2. monk 单测的覆盖盲区：TestUnifiedTreeCoreLoadBackPending 用 mock.Mock 构造 core，只绑定了部分真实方法（node_by_id、_update_duplicate_tracking 等），真实调度器中 _reclaim_full_host_duplicates() 与事件循环、DMA ACK 时序的交互并未在单测中覆盖，依赖 PR 中的 GPT-OSS 手工验证。
  3. ACK 路径开销：finish_load_back() 现在对 write-through 模式也沿 anchor 路径逐节点调用 _update_duplicate_tracking(node)，长 prefix 路径下 ACK 处理会遍历更多节点，带来轻微 CPU 开销；不过该逻辑本来就在异步 ACK 回调路径上，量级有限。
  4. 回归风险：改动集中在一个文件的两个方法，write-back 分支的语义（首个 load-back 钉住、_can_reclaim_full_host_duplicate() 拒绝、ACK 清除）被刻意保持原样，回归风险可控。 - 影响：对使用 --hicache-write-policy write_through 的 HiCache 部署是直接 bugfix：消除了多请求重叠 load-back 时的 AssertionError 崩溃；对 write-back 部署行为不变（仍保持保守的串行化保护）。对系统而言，finish_load_back() 在 write-through 路径上的重复跟踪刷新保证了存储预取场景下 Duplicate missing 不变量检查不误报，为 UnifiedRadixCache 的 Host/GPU 分层缓存一致性提供保障。对团队而言，该改动明确了 load_back_pending_id 的职责边界，为后续 write-back 并发语义的扩展（由 xiezhq-hermann 跟进）划清了范围，属于低风险、高收窄收益的修复。 - 风险

标记：核心缓存路径变更，write-back 并发语义遗留，测试基于 Mock core

关联脉络

- PR #34889 [DCP]Localize HiCache DCP indices once per transfer, not per layer: 同为 HiCache 数据路径优化，涉及 mem_cache 下的 Host pool 与传输索引处理，与本 PR 在同一功能线上（HiCache 分层缓存的正确性与性能收尾）。
- PR #35030 Add bit-exact guard for extra_buffer_lazy: 同为 UnifiedRadixCache 相关的测试与不变量强化，测试文件同属 test/registered/radix_cache/unified_radix_tree 目录体系，关注 radix cache 一致性与空闲检查。