

PR #51787 完整报告

vllm-project/vllm

[Bugfix][KV Offload] Track cache recency once per request

合并时间: 2026-09-16 15:47

原文链接: <http://prhub.com.cn/vllm-project/vllm/pull/51787>

执行摘要

- 一句话: KV Offload 热度改为每请求结算一次, 修正淘汰顺序
- 推荐动作: 建议精读, 尤其是负责 KV Offload、CPU 缓存策略或调度器的工程师:
 - 值得学习的三个设计决策: 一是「用逻辑 rank 而非物理位置表达 recency」, 从而让 pin 状态与淘汰顺序解耦; 二是「每请求一次访问」的语义边界 (insertion / reuse / 内部级联 pin 三分类); 三是通过默认实现保持 touch API 与外部策略兼容的渐进式改法。
 - 重点关注 policies/lru.py 的 `_is_current()` / `_update_recency()` / `evict()` 三者构成的懒失效与原子回滚协议, 这是最容易在后续修改中被破坏的部分。
 - 若维护自定义 `cache_policy_module_path` 策略, 需验证新的 `on_request_finished()` / `on_store_miss()` 默认路径是否符合预期; 若新增 offload key 的产出路径, 务必同时调用 `set_offload_key_position()`, 否则顺序会静默降级。

功能与动机

PR body 明确指出两个问题: 一是卸载调度器在请求解码期间会反复 re-touch 整个已缓存前缀, 既做重复工作, 又让 ARC 这类基于频率的策略把「一个请求」统计成「很多次访问»; 二是被传输 pin 住的 LRU block 会按传输完成顺序重新进入可淘汰集合, 导致前缀头部可能先被淘汰, 留下不可用的缓存后缀。orozerly 在讨论中进一步给出设计方向: OffloadingManager 应利用 `prepare_store` 与 `on_request_finished` 里的 `req_context` 维护 blocks 的 head-to-tail 顺序, 这同时能修正 ARC 因解码期重复 touch 而不准确的频率统计; mindungil 复述方案后 orozerly 回复 Yep! 确认。

实现拆解

1. 摘掉调度器驱动的 touch, 改为记录 token 位置 (`vllm/distributed/kv_transfer/kv_connector/v1/offloading/scheduler.py`)

- 删除 `_touch()` 方法, 并移除 `get_num_new_matched_tokens()` 与 `_build_store_jobs()` 中的两处调用; 原本按 sliding window 裁剪前缀再 touch 的逻辑整体消失。
- `update_offload_keys()` 在生成 offload key 后调用
`self.req_context.set_offload_key_position(key, len(group_state.offload_keys) * group_config.tokens_per_chunk); _make_boundary_key()` 新增 `req_context` 参数并写入 `boundary_tokens`。调度器职责收敛为「声明这个请求观察到了哪些 key、结尾在哪个 token」, 淘汰语义完全下放给 policy。

2. 请求级访问账本 (`vllm/v1/kv_offload/cpu/manager.py`)

- 新增 `_RequestCacheAccess` dataclass (`owner / cache_generation / key_groups / seen_keys / inserted_keys / reused_keys / store_miss_keys / finished`)，经 `ReqContext.set_state()` 挂在请求上；`_get_request_cache_access()` 用 `owner` 与 `_cache_generation` 双重校验，`reset_cache()` 递增 `_cache_generation` 使旧账本失效（对应测试 `test_reset_discards_stale_request_access_classification`）。
- `_record_request_cache_access()` 对 `key` 去重、按 `get_offload_group_idx(key)` 分组，并把本请求刚写入又被读回的 `key` 从 `reused_keys` 中剔除（`tiering cascade` 属于内部搬运，不算第二次访问）。
- `prepare_load()` 拆出 `_prepare_load(..., record_access=...)` 以便内部调用绕过记账；`prepare_store()` 改为先按「是否已在 `cache` 且 `ready`」把原始 `offer` 分区成 `keys_to_store / ready_existing_keys`，再对候选应用 `store_threshold`，`on_store_miss()` 只对首次出现的 `miss` 触发一次策略钩子，`inserted_keys` 仅在实际分配成功后更新。

3. 结算：一次 head-to-tail 逻辑访问 (`manager.on_request_finished + policies/base.order_request_keys`)

- `order_request_keys()` 把所有 `KV group` 的 `key` 展平后按 `ReqContext.get_offload_key_position()` 排序，实现跨 `group` 的全局 `head→tail`（而非仅组内有序）；位置信息不全时保持原顺序，属于静默降级。
- `on_request_finished()` 先置 `state.finished = True`（之后再有新 `key` 观察会触发 `assert`），再把 `inserted_keys - reused_keys` 作为 `insertion_only_keys`、`reused_keys` 作为复用集合传给策略；即使 `load/store` 的传输完成顺序错乱，最终的顺序都由这一处决定。

4. 策略实现：LRU 逻辑 rank + 懒堆，ARC 区分 `insertion / reuse`

- LRU (`policies/lru.py`)：OrderedDict 淘汰列表换成 `_ranks + _evictable + _heap`；`_assign_new_rank()` 允许在 `key` 仍被 `pin` 时更新逻辑 `recency`，`mark_evictable()` 只把当前 `rank` 推入堆；`_is_current()` 用懒失效过滤旧条目，`_maybe_compact_heap()` 与 `_update_recency()` 的长前缀 `rebuild` 保证堆大小有界；`evict()` 采用「先选中、不足则整体回滚（`selected` 与 `deferred` 重新入堆）」的原子语义，并用 `selected_keys` 防止重复选中同一 `key`。
- ARC (`policies/arc.py`)：`_adapt_to_ghost_hit()` 从 `touch()` 抽出，改由 `on_store_miss()` 在 `insert()` 清除 `ghost entry` 之前调用；`on_request_finished()` 中 `insertion-only key` 只做 `move_to_end` 保持成员资格，只有真正 `reuse` 的 `ready chunk` 才 `T1→T2` 晋升。
- `policies/base.py` 为两个新钩子提供默认实现（`on_store_miss` 委托 `touch()`、`on_request_finished` 逐 `group touch`），使 `cache_policy_module_path` 加载的外部策略在不改代码的情况下继续可用。

5. tiering 的跨层语义 (`vllm/v1/kv_offload/tiering/manager.py`、`vllm/v1/kv_offload/base.py`)

- `TieringOffloadingManager.prepare_read()` 不再直接别名到 `prepare_load`，改为显式调用 `_prepare_load(..., record_access=False)`，把 `CPU→secondary` 的级联读标记为内部 `pin`；主 `tier` 的顺序仍在 `on_request_finished` 立即提交，`secondary` 的 `finalization` 继续延迟到不会再有 `cascade` 提交之后。

- ReqContext 新增 `_offload_key_positions` 字段与 `set_offload_key_position()` / `get_offload_key_position()`, 成为 scheduler 与 policy 之间共享的顺序信息来源。

6. 测试与验证配套 | 测试文件 | 增量 | 覆盖点 | | --- | --- | --- | |

`tests/v1/kv_offload/cpu/test_manager.py` | +428 | 请求顺序与 store/load 完成顺序解耦、迟到旧完成不覆盖新请求、LRU 批量淘汰原子性与懒堆压缩、`store_threshold` 不吞掉 reuse、reset 后账本作废、ARC 复用计一次且 insertion 留在 T1 | |

`tests/v1/kv_offload/tiering/test_tiering_offloading.py` | +56 | 级联完成不替换请求顺序、迟到旧 store 提交不算新访问、次级 finalize 延迟语义、touch 顺序断言改为比较 `_ranks` | |

`tests/v1/kv_connector/unit/offloading_connector/test_scheduler.py` | +18/-53 | 断言 touch 不再被调用、断言 `get_offload_key_position()` 记录正确 (含 partial-tail boundary)

I-PR 自述验证结果: 303 个相关单测通过 (50CPUmanager、56tiering、189scheduler、8 policy factory), pre-commit (ruff、mypy) 通过; GPU E2E 用

Qwen2.5-0.5B-Instruct + 64 块 CPU LRU 缓存复现出 768/768 预期缓存 token, 输出与 baseline 一致。

关键文件:

- `vllm/v1/kv_offload/cpu/manager.py` (模块 缓存管理; 类别 source; 类型 core-logic; 符号 `_RequestCacheAccess`, `_get_request_cache_access`, `_record_request_cache_access`, `_prepare_load`): 新增请求级缓存访问账本 `_RequestCacheAccess` 与 `on_request_finished` 结算钩子; `prepare_store` 改为先分区再应用 `store_threshold` 并触发 `on_store_miss`; `reset_cache` 递增 `_cache_generation` 作废旧账本。
- `vllm/v1/kv_offload/cpu/policies/lru.py` (模块 淘汰策略; 类别 source; 类型 core-logic; 符号 `_assign_new_rank`, `_push_evictable`, `_is_current`, `_maybe_compact_heap`): 用逻辑 rank + 懒失效最小堆替换 `OrderedDict` 淘汰列表, 新增 `on_request_finished` 结算与带整体回滚的原子批量 evict。
- `vllm/v1/kv_offload/cpu/policies/arc.py` (模块 自适应缓存; 类别 source; 类型 core-logic; 符号 `_adapt_to_ghost_hit`, `on_store_miss`, `on_request_finished`): 抽出 ghost 命中自适应并交给 `on_store_miss`, 在 `on_request_finished` 中区分 insertion-only 与 reuse, 保证每请求最多计一次 frequency。
- `vllm/v1/kv_offload/cpu/policies/base.py` (模块 策略接口; 类别 source; 类型 core-logic; 符号 `order_request_keys`, `on_store_miss`, `on_request_finished`): 新增跨 group 顺序恢复工具 `order_request_keys`, 以及带默认实现 (委托 touch) 的 `on_store_miss` / `on_request_finished` 钩子, 保持外部策略兼容。
- `vllm/distributed/kv_transfer/kv_connector/v1/offloading/scheduler.py` (模块 卸载调度; 类别 source; 类型 core-logic; 符号 `_touch`, `update_offload_keys`, `_make_boundary_key`, `get_num_new_matched_tokens`): 删除 `_touch` 及两处调用点, 改为在 `update_offload_keys` / `_make_boundary_key` 中通过 `ReqContext.set_offload_key_position` 记录 key 的结束 token 位置。
- `vllm/v1/kv_offload/base.py` (模块 请求上下文; 类别 source; 类型 core-logic; 符号 `set_offload_key_position`, `get_offload_key_position`): ReqContext 新增 `_offload_key_positions` 字段与读写方法, 成为调度器与缓存策略之间共享的顺序信息来源。

- vllm/v1/kv_offload/tiering/manager.py (模块 分层存储; 类别 source; 类型 core-logic; 符号 prepare_read, _maybe_finalize_request) : prepare_read 不再别名到 prepare_load, 改为 _prepare_load(..., record_access=False), 把 CPU→secondary 级联读标记为内部 pin; finalize 语义描述随之更新。
- tests/v1/kv_offload/cpu/test_manager.py (模块 缓存测试; 类别 test; 类型 test-coverage; 符号 test_lru_batch_eviction_failure_is_atomic, test_lru_repeated_pin_cycles_compact_lazy_heap_entries, test_reset_discards_stale_request_access_classification, test_store_threshold_does_not_hide_ready_reuse) : 新增 8 组回归测试覆盖请求级顺序、懒堆压缩、store 阈值不吞掉 reuse、reset 后账本作废与 ARC 分类, 并更新既有 LRU 期望淘汰结果 ([4, 5, 2] → [4, 5, 3]) 。
- tests/v1/kv_offload/tiering/test_tiering_offloading.py (模块 分层测试; 类别 test; 类型 test-coverage; 符号 test_request_order_survives_late_cascade_completion, test_late_old_store_submission_does_not_override_newer_request_order, test_on_request_finished_delays_secondary_until_store_submitted, test_on_request_finished_delays_only_secondary_until_store_submitted) : 新增级联完成顺序与迟到旧 store 提交的回归测试, 并把 touch 顺序断言改写为比较 LRU 的 _ranks。
- tests/v1/kv_connector/unit/offloading_connector/test_scheduler.py (模块 调度测试; 类别 test; 类型 test-coverage; 符号 test_offloading_connector, test_two_groups_full_and_sliding_window, test_two_groups_different_block_sizes, test_partial_tail_store_uses_attention_and_recurrent_cow_sources) : 删除大量 manager.touch 断言并改为 assert_not_called(), 新增对 get_offload_key_position() 记录位置的断言 (含 partial-tail boundary 场景) 。

关键符号: CPUOffloadingManager._get_request_cache_access, CPUOffloadingManager._record_request_cache_access, CPUOffloadingManager._prepare_load, CPUOffloadingManager.prepare_store, CPUOffloadingManager.on_request_finished, CPUOffloadingManager.reset_cache, LRUCachePolicy._assign_new_rank, LRUCachePolicy._is_current, LRUCachePolicy._maybe_compact_heap, LRUCachePolicy._update_recency, LRUCachePolicy.on_request_finished, LRUCachePolicy.evict, ARCCachePolicy._adapt_to_ghost_hit, ARCCachePolicy.on_store_miss, ARCCachePolicy.on_request_finished, order_request_keys, CachePolicy.on_store_miss, CachePolicy.on_request_finished, ReqContext.set_offload_key_position, ReqContext.get_offload_key_position, TieringOffloadingManager.prepare_read, RequestOffloadState.update_offload_keys, OffloadingConnectorScheduler._make_bound ary_key

关键源码片段

vllm/v1/kv_offload/cpu/manager.py

新增请求级缓存访问账本 `_RequestCacheAccess` 与 `on_request_finished` 结算钩子; `prepare_store` 改为先分区再应用 `store_threshold` 并触发 `on_store_miss`; `reset_cache` 递

增 `_cache_generation` 作废旧账本。

```
# vllm/v1/kv_offload/cpu/manager.py
```

```
@dataclass(slots=True)
```

```
class _RequestCacheAccess:
```

```
    """Cache keys observed by one request, grouped in prefix order."""
```

```
    owner: object # 归属的 manager, 避免多个 manager 共用 ReqContext 时串账
```

```
    cache_generation: int # reset_cache() 的代际号, 代际变化即视为旧账本作废
```

```
    key_groups: dict[int, list[OffloadKey]] = field(default_factory=dict)
```

```
    seen_keys: set[OffloadKey] = field(default_factory=set) # 去重: 同一 key 每请求只算一次
```

```
    inserted_keys: set[OffloadKey] = field(default_factory=set) # 本请求写入的 key
```

```
    reused_keys: set[OffloadKey] = field(default_factory=set) # 本请求真正复用的 ready key
```

```
    store_miss_keys: set[OffloadKey] = field(default_factory=set)
```

```
    finished: bool = False
```

```
def _record_request_cache_access(
```

```
    self,
```

```
    keys: Iterable[OffloadKey],
```

```
    req_context: ReqContext,
```

```
    inserted_keys: Iterable[OffloadKey] = (),
```

```
    reused_keys: Iterable[OffloadKey] = (),
```

```
) -> None:
```

```
    state = self._get_request_cache_access(req_context)
```

```
    for key in keys:
```

```
        if key in state.seen_keys: # 同一请求内重复出现不重复计数
```

```
            continue
```

```
        assert not state.finished, "New cache keys observed after request finalization"
```

```
        group_idx = get_offload_group_idx(key)
```

```
        state.key_groups.setdefault(group_idx, []).append(key)
```

```
        state.seen_keys.add(key)
```

```
    state.inserted_keys.update(inserted_keys)
```

```
    # 本请求刚写入又被读回的 key 属于内部搬运 (如 tiering 级联), 不算第二次缓存访问
```

```
    state.reused_keys.update(
```

```
        key for key in reused_keys if key not in state.inserted_keys
```

```
    )
```

```
@override
```

```
def on_request_finished(self, req_context: ReqContext) -> None:
```

```
    """请求结束时一次性提交本请求的逻辑访问顺序, 之后不再接受新 key."""
```

```
    state = req_context.get_state(_RequestCacheAccess)
```

```
    if (
```

```
        state is None
```

```
        or state.owner is not self
```

```
        or state.cache_generation != self._cache_generation
```

```
        or state.finished
```

```

):
    return
state.finished = True
key_groups = []
for group_idx in sorted(state.key_groups):
    keys = state.key_groups[group_idx]
    positions = {
        key: position
        for key in keys
        if (position := req_context.get_offload_key_position(key)) is not None
    }
    if len(positions) == len(keys):
        keys = sorted(keys, key=positions.__getitem__) # 按结束 token 位置恢复 head→tail
    key_groups.append(tuple(keys))
self._policy.on_request_finished(
    tuple(key_groups),
    state.inserted_keys - state.reused_keys, # insertion-only: 写入不构成 frequency hit
    state.reused_keys, # 真正被复用的 ready chunk 才计一次访问
    req_context,
)

```

vllm/v1/kv_offload/cpu/policies/lru.py

用逻辑 rank + 懒失效最小堆替换 OrderedDict 淘汰列表，新增 `on_request_finished` 结算与带整体回滚的原子批量 `evict`。

```
# vllm/v1/kv_offload/cpu/policies/lru.py
```

```

def _assign_new_rank(self, key: OffloadKey) -> bool:
    """key 仍被 pin 时也能更新逻辑 recency，与传输完成顺序彻底解耦。"""
    if key not in self.chunks:
        return False
    self._next_rank += 1
    self._ranks[key] = self._next_rank
    return key in self._evictable # 只有可淘汰的 key 才需要把新 rank 推进堆

```

```

def _push_evictable(self, key: OffloadKey) -> None:
    heapq.heappush(self._heap, (self._ranks[key], key))

```

```

def _is_current(self, entry: tuple[int, OffloadKey]) -> bool:
    """懒失效：堆里可能残留旧 rank 的条目，rank 不一致直接丢弃。"""
    rank, key = entry
    return key in self._evictable and self._ranks.get(key) == rank

```

```

def _maybe_compact_heap(self) -> None:
    """堆长度超过 max(64, 2 × 可淘汰数) 时整体重建，保证内存有上界。"""
    if len(self._heap) < max(64, 2 * len(self._evictable)):

```

```

        return
    self._heap = [(self._ranks[key], key) for key in self._evictable]
    heapq.heapify(self._heap)

def _update_recency(self, keys: Iterable[OffloadKey]) -> None:
    updated_evictable = [key for key in keys if self._assign_new_rank(key)]
    # 长前缀整体重建是线性开销，明显便宜于逐个 heappush；小批量仍走增量路径
    if len(updated_evictable) >= max(64, len(self._evictable) // 4):
        self._heap = [(self._ranks[key], key) for key in self._evictable]
        heapq.heapify(self._heap)
    else:
        for key in updated_evictable:
            self._push_evictable(key)
        self._maybe_compact_heap()

@override
def on_request_finished(
    self,
    key_groups: Sequence[Sequence[OffloadKey]],
    insertion_only_keys: set[OffloadKey],
    reused_keys: set[OffloadKey],
    req_context: ReqContext,
) -> None:
    del insertion_only_keys, reused_keys # LRU 不区分插入与复用，只关心顺序
    # 从 tail 到 head 依次赋 rank: head 最后赋值 → rank 最大 → 最晚被淘汰
    self._update_recency(reversed(order_request_keys(key_groups, req_context)))

```

vllm/v1/kv_offload/cpu/policies/arc.py

抽出 ghost 命中自适应并交给 `on_store_miss`，在 `on_request_finished` 中区分 insertion-only 与 reuse，保证每请求最多计一次 frequency。

```

# vllm/v1/kv_offload/cpu/policies/base.py
# 跨 hybrid KV group 恢复统一的 head→tail 顺序，供各策略在结算时复用
def order_request_keys(
    key_groups: Sequence[Sequence[OffloadKey]], req_context: ReqContext
) -> list[OffloadKey]:
    """Return one head-to-tail order across all KV cache groups."""
    keys = [key for group in key_groups for key in group]
    positions = {
        key: position
        for key in keys
        if (position := req_context.get_offload_key_position(key)) is not None
    }
    if len(positions) == len(keys):
        keys.sort(key=positions.__getitem__) # 位置信息不全时保持原顺序（静默降级）
    return keys

```

```

# vllm/v1/kv_offload/cpu/policies/arc.py
@override
def on_store_miss(self, keys: Iterable[OffloadKey], req_context: ReqContext) -> None:
    """在 insert() 抹掉 ghost entry 之前，每请求最多自适应一次 target_t1_size。"""
    for key in reversed(list(keys)):
        self._adapt_to_ghost_hit(key)

@override
def on_request_finished(
    self,
    key_groups: Sequence[Sequence[OffloadKey]],
    insertion_only_keys: set[OffloadKey],
    reused_keys: set[OffloadKey],
    req_context: ReqContext,
) -> None:
    for key in reversed(order_request_keys(key_groups, req_context)):
        if key in insertion_only_keys:
            # store 写入本身不算 frequency hit: 保留 T1/T2 成员资格，只恢复 tail→head 的 recency
            if key in self.t1:
                self.t1.move_to_end(key)
            elif key in self.t2:
                self.t2.move_to_end(key)
            continue

        # 只有本请求真正复用的 ready chunk 才算一次访问: T1 → T2 晋升, T2 移到 MRU
        if key in reused_keys and key in self.t1:
            chunk = self.t1.pop(key)
            self.t2[key] = chunk
        elif key in reused_keys and key in self.t2:
            self.t2.move_to_end(key)

```

评论区精华

- 设计定调 (orozery, issue 评论): The OffloadingManager should use the req_context in prepare_store and in on_request_finished to maintain the order of blocks heads to tail. This will also solve the current issue with frequency based evictions (such as ARC) which currently are inaccurate due to the fact that keys are re-touched while the request is decoding.
- 方案复述与确认: mindungil 提出「移除调度器 touch、用 ReqContext 记录 load/store 访问、在 on_request_finished 提交 head-to-tail 顺序、ARC 常驻块每请求计一次、新写入留在 T1」, orozery 回复 Yep! 直接确认, 说明本次改动是双方事先对齐过的设计, 而非事后修补。
- 正确性交锋 (coderabbitai vs mindungil): coderabbitai 指出当 store_threshold >= 2 时, _record_accesses() 会把老化的常驻 key 计数重置为 1, 随后被阈值过滤掉, 导致该 key 既不进入 reused_keys 也不进入 on_request_finished, recency 保持陈旧, LRU/ARC 可能淘汰仍可复用的前缀。mindungil 在 49f5cc0e 修复: 先分区原始 offer, 阈

值只作用于 `keys_to_store`，并补 `test_store_threshold_does_not_hide_ready_reuse` 回归测试。

- 可维护性意见 (orozery, 引用 Claude 的评审) : Four separate `_record_request_cache_access` call sites in this method (empty store, eviction impossible, eviction failed, success), each with slightly different args. Missing one on a future early-return silently breaks per-request accounting. → mindungil 在 db68d38d58 改为任何早退之前统一记录一次，分配成功后再更新 `inserted_keys`，并补失败分配路径的测试。
- 命名 (orozery) : Let us rename `touch` to `on_store_miss` to make it more explicit. → 新增独立 `CachePolicy.on_store_miss()` 钩子承载 ARC 的 ghost 自适应，`touch` 保留给缓存访问语义与外部策略兼容。
- 边界划分 (Change72) : Yes, this aligns well with #49413. Event work can reuse the key-position mapping in `ReqContext`. The scopes are complementary: this PR handles cache recency and eviction, while #49413 handles event provenance and removal.
- 流程层面: CI 前经历多次 rebase (曾落后 main 170 个提交)，最终 Buildkite #89264 触发并通过，orozery 以 Thanks @mindungil! 审批。
- 移除调度器 `touch`、改为请求结束一次性结算 (design): 方案确认并按此实现: 删除 `_touch` 与两处调用点、引入 `_RequestCacheAccess` 账本、在 `on_request_finished` 提交一次 head→tail 访问。
- ready resident key 必须在 `store_threshold` 过滤前分类 (correctness): mindungil 在 49f5cc0e 修复: 先对原始 offer 分区出 `keys_to_store / ready_existing_keys`，阈值只作用于候选，并新增 `test_store_threshold_does_not_hide_ready_reuse` 回归测试。
- `prepare_store` 中 4 处 `_record_request_cache_access` 调用点收敛为单点 (design): mindungil 在 db68d38d58 改为任何与分配结果相关的早退之前统一记录一次，分配成功后再更新 `inserted_keys`，并补失败分配路径的回归测试。
- 把 `store` 路径的 `touch` 重命名为 `on_store_miss` (design): db68d38d58 新增独立 `CachePolicy.on_store_miss()` 钩子承载 ARC 的 ghost 自适应，`touch` 保留给缓存访问语义并作为外部策略的兼容回退。
- 与 #49413 (事件溯源 / 移除) 的边界划分 (design): 确认 `ReqContext` 的 key-position 映射可被后续事件工作复用，两个 PR 无冲突、边界清晰。

风险与影响

- 风险:
 1. 记账路径的人工调用风险 (中) : `_record_request_cache_access()` 与 `on_request_finished()` 仍靠调用方显式触发，`order_request_keys()` 在位置信息不全时静默退回原顺序而不报错；若未来新增 `store/load` 分支忘记记账或忘记 `finalize`，淘汰顺序会静默退化且难以察觉 (`assert not state.finished` 只能挡住结算后写入)。
 2. 语义边界收窄 (中) : 只有被 offloading manager 观察到的 key 才影响 CPU 缓存 recency，纯 GPU 本地命中、推测式 `lookup()`、新写入、tiering 级联读都不再是 frequency hit。依赖旧「解码期持续 `touch`」语义的下游 (外部策略、调优经验、容量规

划) 需要重新校准。

3. 外部策略兼容性 (中) : CachePolicy 新增 `on_store_miss()` / `on_request_finished()`, 默认实现分别委托 `touch()` 与逐 group touch, 因此 out-of-tree 策略 (`cache_policy_module_path`) 的行为从「每步 touch」变成「请求结束一次性 touch」, ARC 原本的 `insertion→T2` 晋升路径也被移除; 这类策略需要人工复核。
4. LRU 懒堆的内存与时延权衡 (低 - 中) : 每次 recency 更新可能 push 新条目, 靠 `_maybe_compact_heap()` (阈值 `max(64, 2 * len(_evictable))`) 和 `_update_recency()` 的长前缀整体重建控制; `test_lru_repeated_pin_cycles_compact_lazy_heap_entries` 只覆盖 1 个 key、128 次 pin 的场景, 长前缀高并发下的 rebuild 与 heapify 开销 ($O(n)$, 位于请求结束路径) 缺少压测数据。
5. 淘汰结果变化会扩散到其它测试 (低) : `test_cpu_manager` 的 `evicted_keys` 从 `[4, 5, 2]` 改为 `[4, 5, 3]`, `test_scheduler.py` 大量删除 touch 断言; 任何依赖具体淘汰顺序的测试、benchmark 或外部回归脚本都需要同步更新。

• 影响:

- 线上行为: KV Offload 命中前缀不再因为「头部先被淘汰」而整体失去可用后缀, 减少重复 prefill; ARC 的频率统计不再被单请求放大, 长时间运行下缓存构成更稳定。PR 作者明确该改动只影响 CPU-cache 淘汰记账, 模型评估为 N/A。
- 系统契约: CPUOffloadingManager 与 CachePolicy 之间的接口扩大 (新增 `on_store_miss()`、`on_request_finished()`), ReqContext 新增 key→结束 token 位置的映射, 属于跨模块契约变更; 卸载调度器则不再持有任何缓存策略语义, 职责更清晰。
- 性能: decode 步内对整段前缀的重复 touch 被移除, 减少热路径重复工作; 代价是每个请求结束时多一次排序 + 策略结算 ($O(n \log n)$ 级), 以及 LRU 堆的懒条目管理。
- 团队协作: 与 #49413 的事件溯源 / 移除工作共享 ReqContext 的 key-position 映射, 成为后续 offload 事件管理的基础设施; 外部缓存策略维护者需要跟进新钩子语义。
- 风险标记: 核心淘汰路径变更, 外部策略语义变化, 顺序缺失时静默降级, 懒堆内存与时延开销, 测试期望随行为扩散

关联脉络

- PR #56709 [Bugfix][KV Offload] Restore MTP-retained sliding-window history: 改动同一组文件 `vllm/distributed/kv_transfer/kv_connector/v1/offloading/scheduler.py` 与 `tests/v1/kv_connector/unit/offloading_connector/test_scheduler.py`, 同属卸载调度器前缀 / history 语义修复脉络; 本 PR 删除的 `_touch` 正是该文件的核心逻辑。
- PR #55885 [KV Offload] Add per-request max_load_tokens control: 同样修改 offloading scheduler 与其单测 (`_lookup_complete_chunks` 中 `max_load_tokens` 的收敛逻辑), 说明卸载调度器的阻塞 / 加载窗口一直在被逐项修正, 本 PR 接续这一演化并改写了同一测试文件的断言。
- PR #55055 [BugFix][Model Runner V2][Spec Decode] Fix decode instance's multi-layer MTP kv caches during P/D: 同属 KV cache 正确性修复 (kv-connector / speculative-decoding), 与本 PR 一样处理「异步传输完成与请求状态不一致」的问题, 是同一类跨模块时序缺陷的邻居。

- PR #54014 [Bugfix][KV Offload] Check cgroup memory before SHM allocation: 同为 KV Offload 基础设施方向的修复（涉及 vllm/v1/kv_offload/cpu/ 与 SHM 分配），说明该子系统近期处于密集加固期，本 PR 属于其中的缓存语义一环。