Project 3 - Understanding LRU-IPV

1 a. **What is the LRU replacement policy? Why do we need a replacement policy for a set-associative cache and a fully associative cache?**

The Least-Recently Used (LRU) strategy always removes the block that has been inactive for the longest period of time, keeping account of the sequence in which cache blocks were last read.

• Every cache set keeps track of its lines in a recency stack, with position 0 denoting Most Recently Used (MRU) and position k–1 denoting Least Recently Used (LRU).

• The accessed block is shifted to MRU upon a hit.

• On a miss, the existing LRU block is evicted and a new block is inserted at MRU.

The replacement policy determines which existing block should be replaced by the cache when all paths are full and a new block needs to be inserted.

(LRU predicts future reuse based on "recency of use.")

Because a memory address might map to more than one slot (way) inside a set, a replacement strategy is required in a set-associative or fully-associative cache.

b. As discussed in the paper, why does the LRU replacement policy need an improvement? Explain the concept of insertion/promotion vector and how it improves the LRU replacement policy.

Plain LRU always inserts new blocks as MRU (the hottest position) because it anticipates that each new block will be reused quickly.

This is inefficient in big last-level caches (LLCs) because a high number of fetched blocks are "dead-on-arrival", that is, they are never accessed again.

Until they drift down to LRU, these dead lines consume cache space and remain close to the top of the stack.

The Insertion/Promotion Vector (IPV), a tiny table that specifies where blocks should be inserted on a miss and how far they should be promoted on a hit, is introduced in the article to improve this:

• In a k-way cache, the insertion location for a new line is given by V[k], while the new position following a hit at position i is given by V[0..k-1].

• For instance, LRU = all zeros (always advances to MRU); an additional IPV may insert at LRU or only partially promote.

By fine-tuning these locations (discovered automatically using a genetic approach), the cache reduces misses and boosts speed without requiring additional hardware by keeping dead lines cold and swiftly evicting them while rewarding hot lines with promotion.

c. Based on your understanding of the paper, explain the key ideas behind the tree based PseudoLRU with the help of the algorithms provided in Figures 5 and 6 and Section 3.1.

Tree-based PseudoLRU (PLRU) uses significantly less bits to approximate LRU.

Every internal node contains one PLRU bit that indicates whether side (left or right) was used most recently. Each set is represented as a binary tree, with the cache methods as its leaves.

• Selection of victims (Figure 5):

Beginning at the root, proceed down the coldest branch at each node until you reach a leaf, which is the victim. If bit = 0 indicates that the left was recent, go right; if bit = 1 indicates that the right was recent, continue left.

• On a cache hit, flip the bits on that path so they now point towards the accessed child (designating that side as recently utilised) by walking upward from the accessed leaf (Figure 6).

Near-LRU performance is achieved at a fraction of the cost using this design, which requires just k–1 bits per set (15 bits for 16-way) and updates at most $\log_2 k$ bits each access.

d. Based on Figure 7 and 8, and Section 3.2, which describes the fnd_index(p) algorithm to compute the position of a block p which is the leaf node, in the PseudoLRU recency stack. Explain how the algorithm uses the PLRU bits and the structure of the binary tree to determine the recency position of p.

The PLRU tree's leaf block p's recency position is determined by the function find_index(p). It reads the PLRU bit for each parent as it ascends from p to the root:

• The complement, or opposite, of the parent's bit is appended if p is the left child.

• The bit is appended exactly as is if p is the correct child.The binary number created from the gathered bits indicates the "coldness" of p: bigger values indicate colder (LRU), and smaller numbers indicate more recent (MRU).

Example (4-way): If the bits are A = 0 (root), B = 1, and C = 0, then bits "00" → index 0 (MRU) are produced by Way 3 (right-right) while bits "11" → index 3 (LRU) are produced by Way 1 (left-right).

Consequently, find_index() converts the bit pattern of the tree into a numerical recency

position.

e. Based on Sections 2.4 and 2.5, answer this question. Figure 3 shows the transition graph for an insertion/promotion vector - [0 0 1 0 3 0 1 2 1 0 5 1 0 0 1 11 13]. Explain how the transition graph in Figure 3 differs from the LRU transition graph shown in Figure 2.

• Every node (position 0…k-1) in Figure 2 (the LRU graph) has a strong edge to 0 → on every hit, the block advances to MRU. Moreover, insertions go to 0. The graph is monotonic and consistent.

• The learnt IPV graph in Figure 3: Depending on the position, edges move to different targets: new insertions go close to LRU (position 13), some hits promote a line all the way to MRU (position 5 to 0), and others only halfway (position 4 to 3).

Because of the non-uniform and non-monotonic nature of the graph, likely-dead lines have shorter lifetimes while reused lines have longer lifetimes.

In contrast to LRU's one-size-fits-all behaviour, Figure 3's transition graph displays varying insertion and promotion routes that bias the cache towards retaining valuable data and evicting dead data more quickly.

2. Analyze the code and logic for the implementation of the functions (reset(), touch(), invalidate(), getVictim()) and replacement policies from the files as mentioned. Based on your understanding, answerthe following questions.

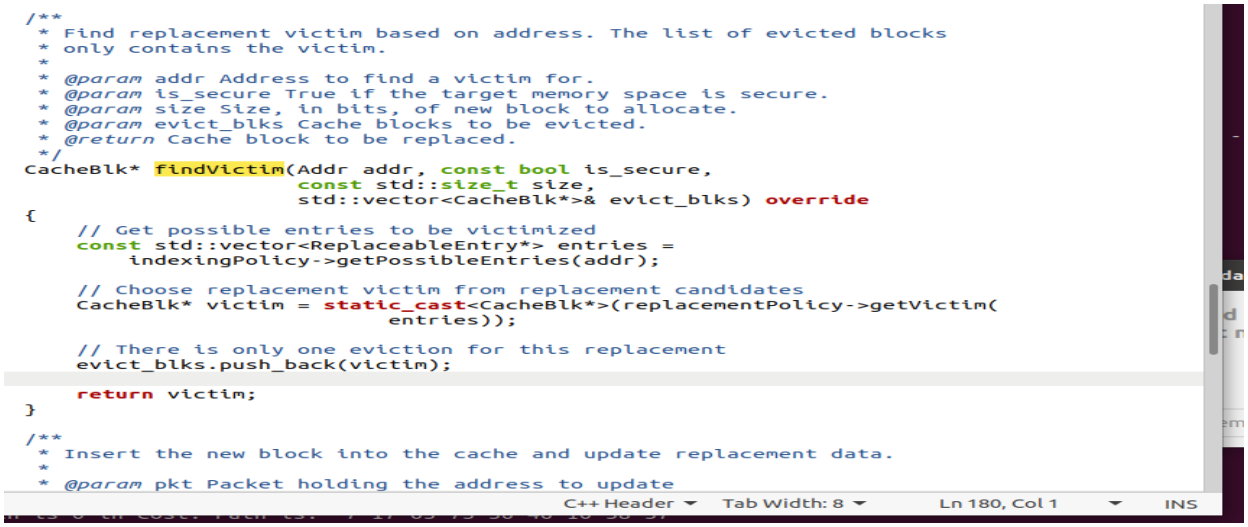
a.[0.4pt] Based on your understanding after reading base_set_assoc.cc and base_set_assoc.hh, explain the specific roles of invalidate() and findVictim() functions in terms of cache management.

1. findVictim() — Choosing a line to replace

Purpose: Selects the cache block to be evicted when a miss occurs.

Where it fits:

- Invoked during miss handling by the higher-level cache logic (BaseTags::accessBlock → findVictim).
- Uses the replacement policy (e.g., LRU) to pick one candidate among all blocks in the indexed set.



```
/**
 * Find replacement victim based on address. The list of evicted blocks
 * only contains the victim.
 *
 * @param addr Address to find a victim for.
 * @param is_secure True if the target memory space is secure.
 * @param size Size, in bits, of new block to allocate.
 * @param evict_blks Cache blocks to be evicted.
 * @return Cache block to be replaced.
 */
CacheBlk* findVictim(Addr addr, const bool is_secure,
                     const std::size_t size,
                     std::vector<CacheBlk*>& evict_blks) override
{
    // Get possible entries to be victimized
    const std::vector<ReplaceableEntry*> entries =
         indexingPolicy->getPossibleEntries(addr);

    // Choose replacement victim from replacement candidates
    CacheBlk* victim = static_cast<CacheBlk*>(replacementPolicy->getVictim(
                            entries));

    // There is only one eviction for this replacement
    evict_blks.push_back(victim);

    return victim;
}

/**
 * Insert the new block into the cache and update replacement data.
 *
 * @param pkt Packet holding the address to update
```

C++ Header Tab Width: 8 Ln 180, Col 1 INS

Explanation:

1. getPossibleEntries(addr) returns all blocks that belong to the same set.
2. replacementPolicy->getVictim(entries) invokes the policy's getVictim() method (e.g., LRU chooses the block with the smallest timestamp).
3. The chosen victim is pushed into evict_blks and returned for eviction.

Conceptually:

findVictim() acts as the bridge between the set's block list and the replacement policy. It doesn't decide the victim itself; instead, it delegates that decision to the selected policy (LRU, PLRU, LRU-IPV, etc.). This modular design lets gem5 swap different policies easily.\

2. invalidate() — Marking a block invalid

Purpose: Updates the tag store and replacement-policy metadata when a block becomes invalid (e.g., on replacement or coherence invalidation).

Explanation:

1. BaseTags::invalidate(blk) clears the valid bit in the tag array.
2. stats.tagsInUse-- decrements the counter tracking active cache lines.
3. replacementPolicy->invalidate() notifies the replacement policy to update its metadata (e.g., mark the entry as least-recently-used or unusable).

Conceptually:
invalidate() keeps the replacement-policy metadata synchronized with the actual cache contents.
When a line is invalidated, its metadata is also cleared or aged, making it the first choice for the next insertion.

3. Relation to Cache Management

Cache Event	**Function Called**	**Metadata Update**	**Effect on Block**
Insertion (miss)	reset()	lastTouchTick = curTick()	Block becomes MRU

Cache Event	Function Called	Metadata Update	Effect on Block
Hit	touch()	lastTouchTick = curTick()	Block stays MRU
Invalidation	invalidate()	lastTouchTick = 0	Block becomes oldest
Replacement	getVictim()	Scans all timestamps, chooses minimum	Evicts LRU block

4. Interaction with Replacement Policy

Both functions directly use the replacementPolicy pointer:

- findVictim() → delegates *selection* to getVictim() (e.g., LRU's timestamp comparison).
- invalidate() → delegates *metadata update* to invalidate() in the policy class (e.g., set lastTouchTick = 0 for LRU).

This separation allows gem5 to support many replacement policies (LRU, tree-PLRU, random, LRU-IPV, etc.) under a common interface.

In base_set_assoc.cc and .hh, the findVictim() function is responsible for identifying which cache block within a set should be evicted upon a miss. It gathers all candidate entries using indexingPolicy->getPossibleEntries() and asks the active replacement policy through getVictim() to choose one based on its metadata (e.g., LRU timestamps).

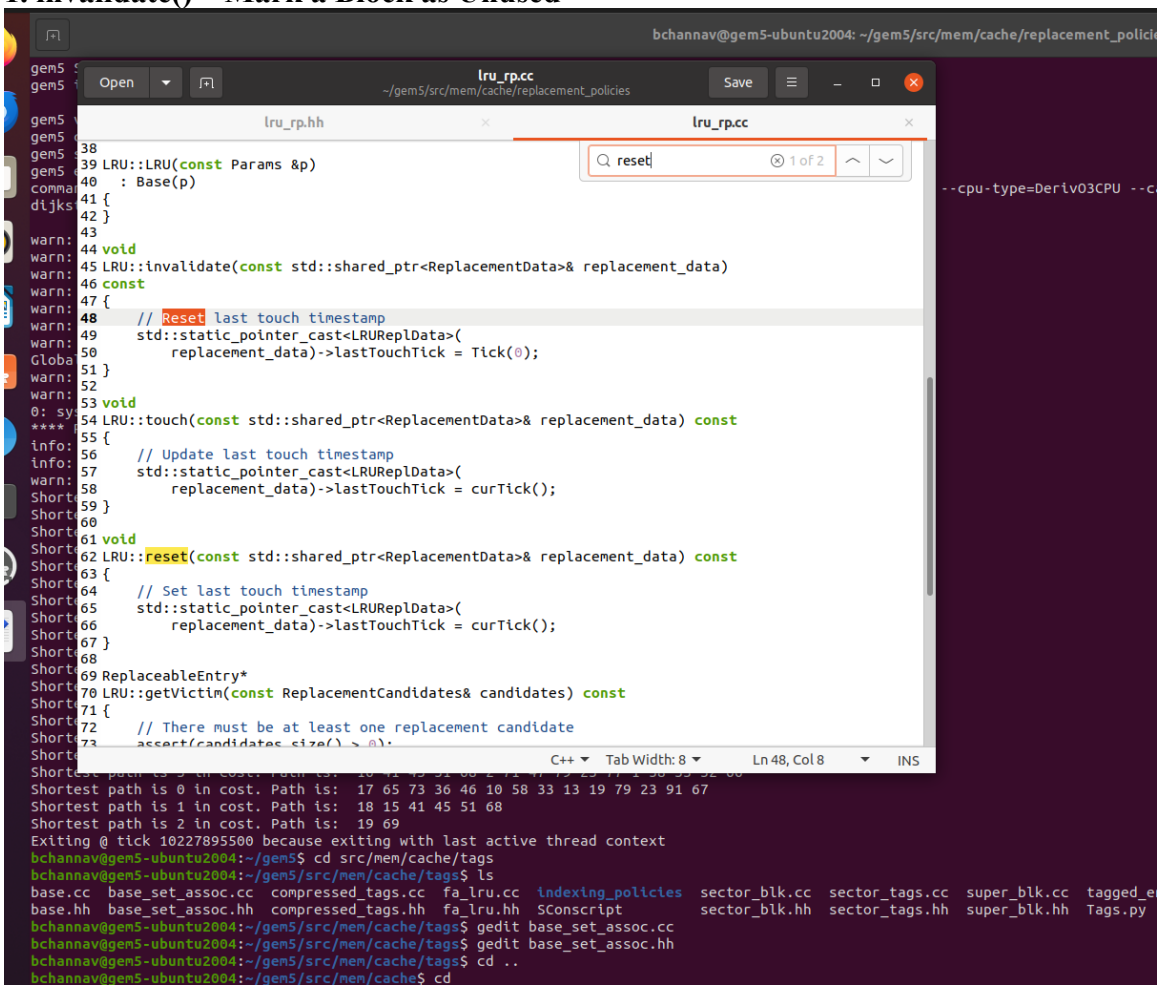
The invalidate() function is invoked whenever a block becomes invalid—either during replacement or by coherence actions. It marks the tag as invalid, updates usage statistics, and calls replacementPolicy->invalidate() to synchronize the metadata so the block becomes the next candidate for reuse.

Together, these two functions form the core replacement mechanism of the cache: findVictim() decides which block to remove, while invalidate() updates the internal state to keep the replacement policy consistent with the cache contents.

b.[0.6pt] The lru_rp.cc and lru_rp.hh files implement LRU (Least Recently Used) cache replacement policy. Based on your understanding of the implementation, explain the specific roles of each of the following functions in implementing the LRU policy: reset(), touch(), invalidate(), and getVictim().

The LRU (Least Recently Used) replacement policy in gem5 tracks when each cache block was last accessed using a variable called 'lastTouchTick'. The class LRU is defined in src/mem/cache/replacement_policies/lru_rp.hh and implemented in lru_rp.cc. It derives from BaseReplacementPolicy and overrides four key functions that correspond to cache management events: invalidate(), touch(), reset(), and getVictim(). Each function updates or uses the 'lastTouchTick' metadata so that the policy always knows which block was least recently used.

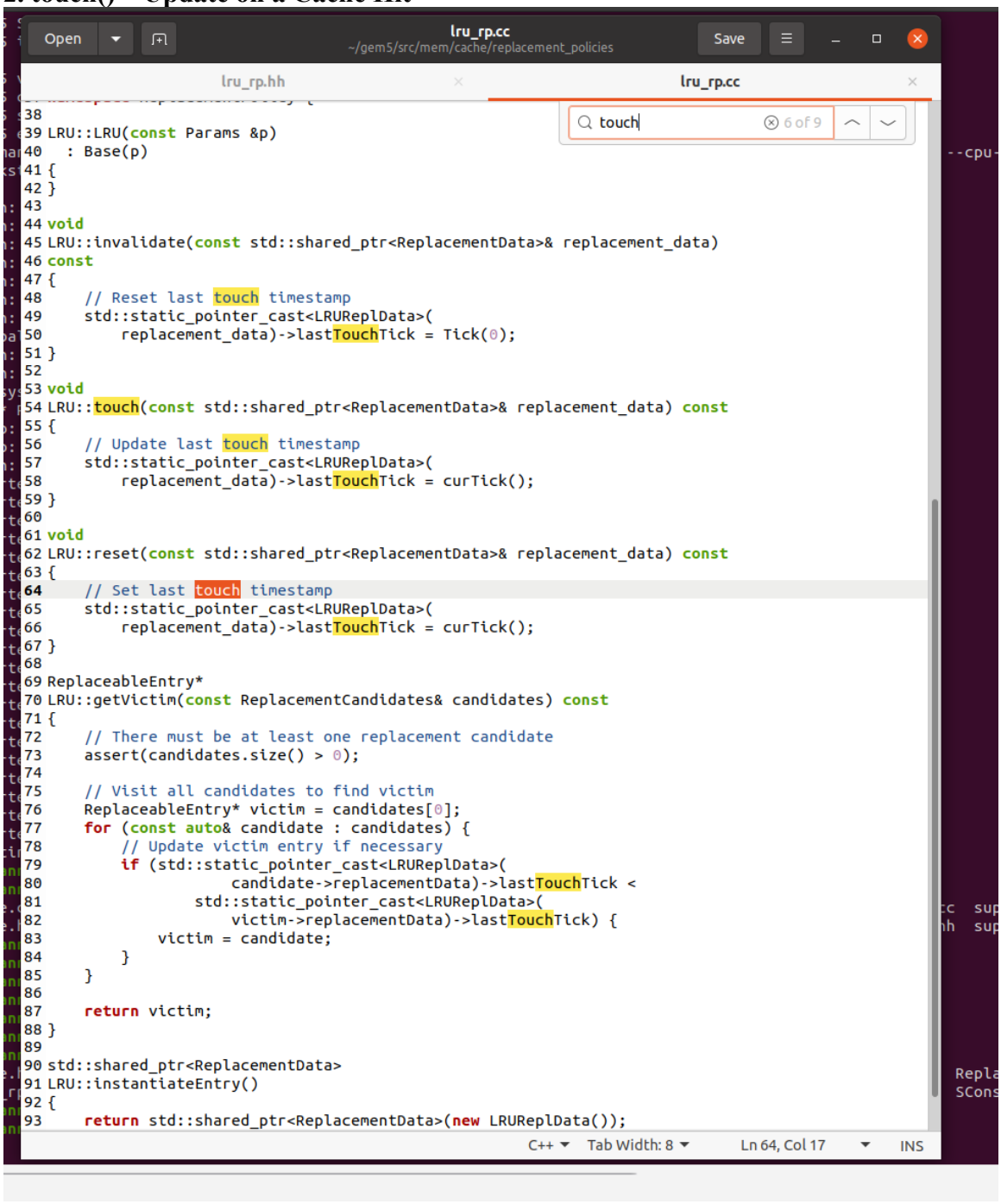
1. invalidate() – Mark a Block as Unused



Description:
- Called when a cache block is invalidated for example, when it is replaced or coherently invalidated.
- Sets its lastTouchTick to 0, the lowest possible value.
- This makes the block appear oldest, so it is the first candidate for eviction during the next replacement.

Purpose in Cache Management:
Ensures that invalid blocks become eligible for reuse and synchronizes the replacement metadata with the actual cache state.

2. touch() – Update on a Cache Hit

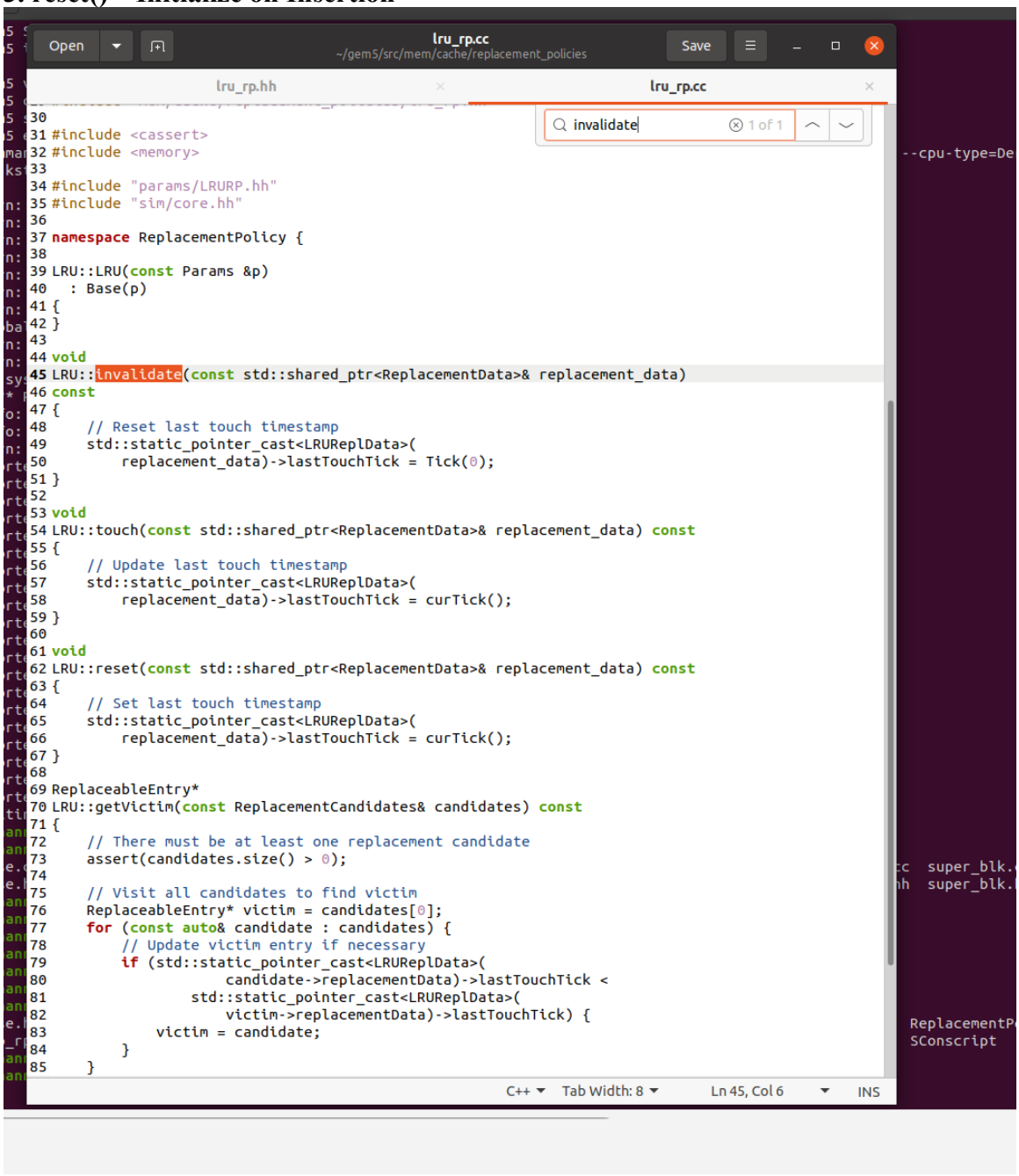



```
38
39 LRU::LRU(const Params &p)
40   : Base(p)
41 {
42 }
43
44 void
45 LRU::invalidate(const std::shared_ptr<ReplacementData>& replacement_data)
46 const
47 {
48     // Reset last touch timestamp
49     std::static_pointer_cast<LRUReplData>(
50         replacement_data)->lastTouchTick = Tick(0);
51 }
52
53 void
54 LRU::touch(const std::shared_ptr<ReplacementData>& replacement_data) const
55 {
56     // Update last touch timestamp
57     std::static_pointer_cast<LRUReplData>(
58         replacement_data)->lastTouchTick = curTick();
59 }
60
61 void
62 LRU::reset(const std::shared_ptr<ReplacementData>& replacement_data) const
63 {
64     // Set last touch timestamp
65     std::static_pointer_cast<LRUReplData>(
66         replacement_data)->lastTouchTick = curTick();
67 }
68
69 ReplaceableEntry*
70 LRU::getVictim(const ReplacementCandidates& candidates) const
71 {
72     // There must be at least one replacement candidate
73     assert(candidates.size() > 0);
74
75     // Visit all candidates to find victim
76     ReplaceableEntry* victim = candidates[0];
77     for (const auto& candidate : candidates) {
78         // Update victim entry if necessary
79         if (std::static_pointer_cast<LRUReplData>(
80                     candidate->replacementData)->lastTouchTick <
81                 std::static_pointer_cast<LRUReplData>(
82                     victim->replacementData)->lastTouchTick) {
83             victim = candidate;
84         }
85     }
86
87     return victim;
88 }
89
90 std::shared_ptr<ReplacementData>
91 LRU::instantiateEntry()
92 {
93     return std::shared_ptr<ReplacementData>(new LRUReplData());
```

Description:
- Invoked whenever a cache hit occurs on a block.
- Updates the block's lastTouchTick to the current simulation tick (curTick()).
- This marks it as the most recently used (MRU) block.

Purpose in Cache Management:
Keeps the recency ordering accurate by refreshing the access timestamp on each hit, preventing frequently used blocks from being evicted.

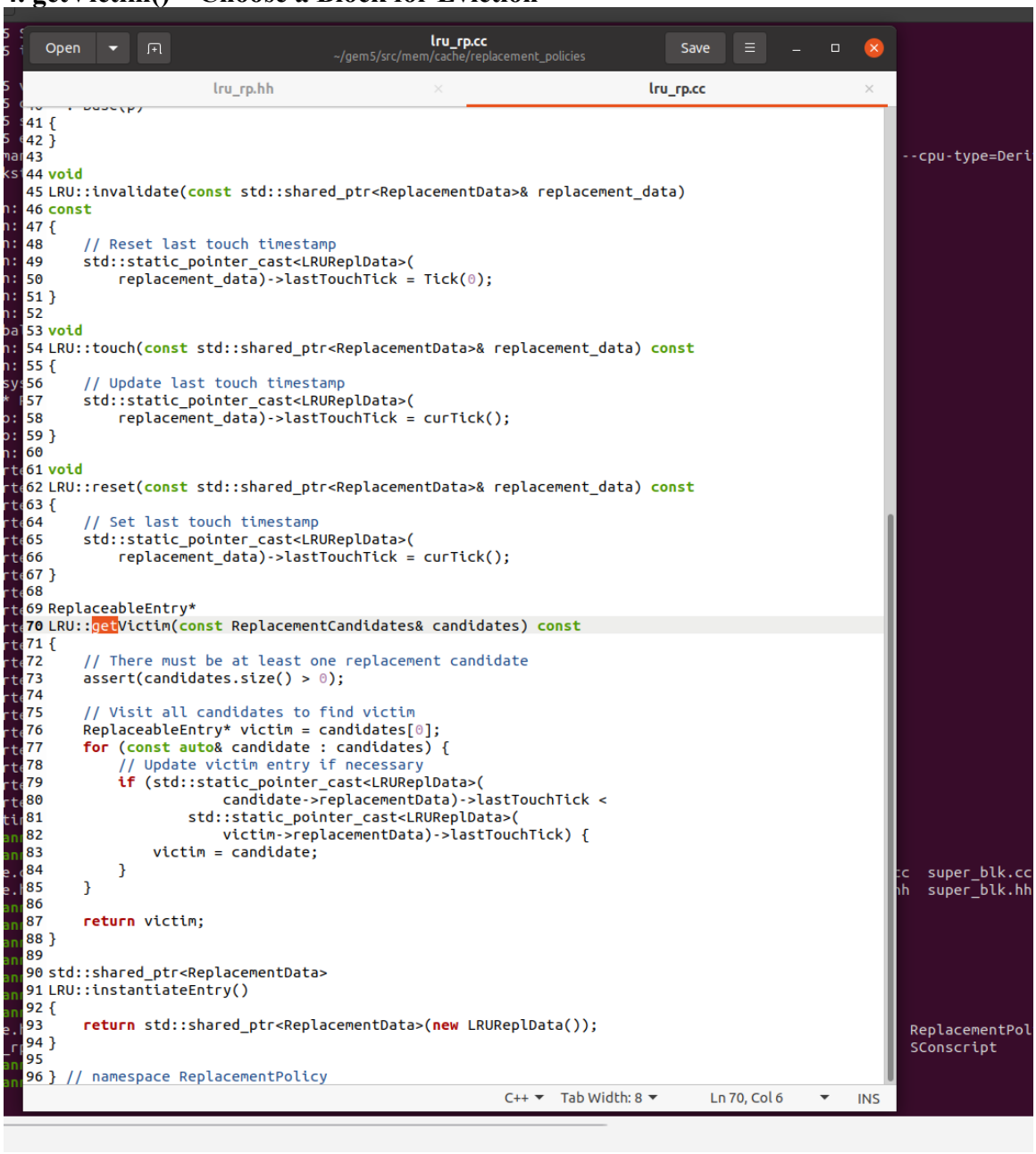
3. reset() – Initialize on Insertion



Description:
- Called when a new cache block is inserted (after a miss and eviction).
- Initializes its metadata so that the block is considered freshly used.
- Uses the same logic as touch(), setting the timestamp to the current tick.

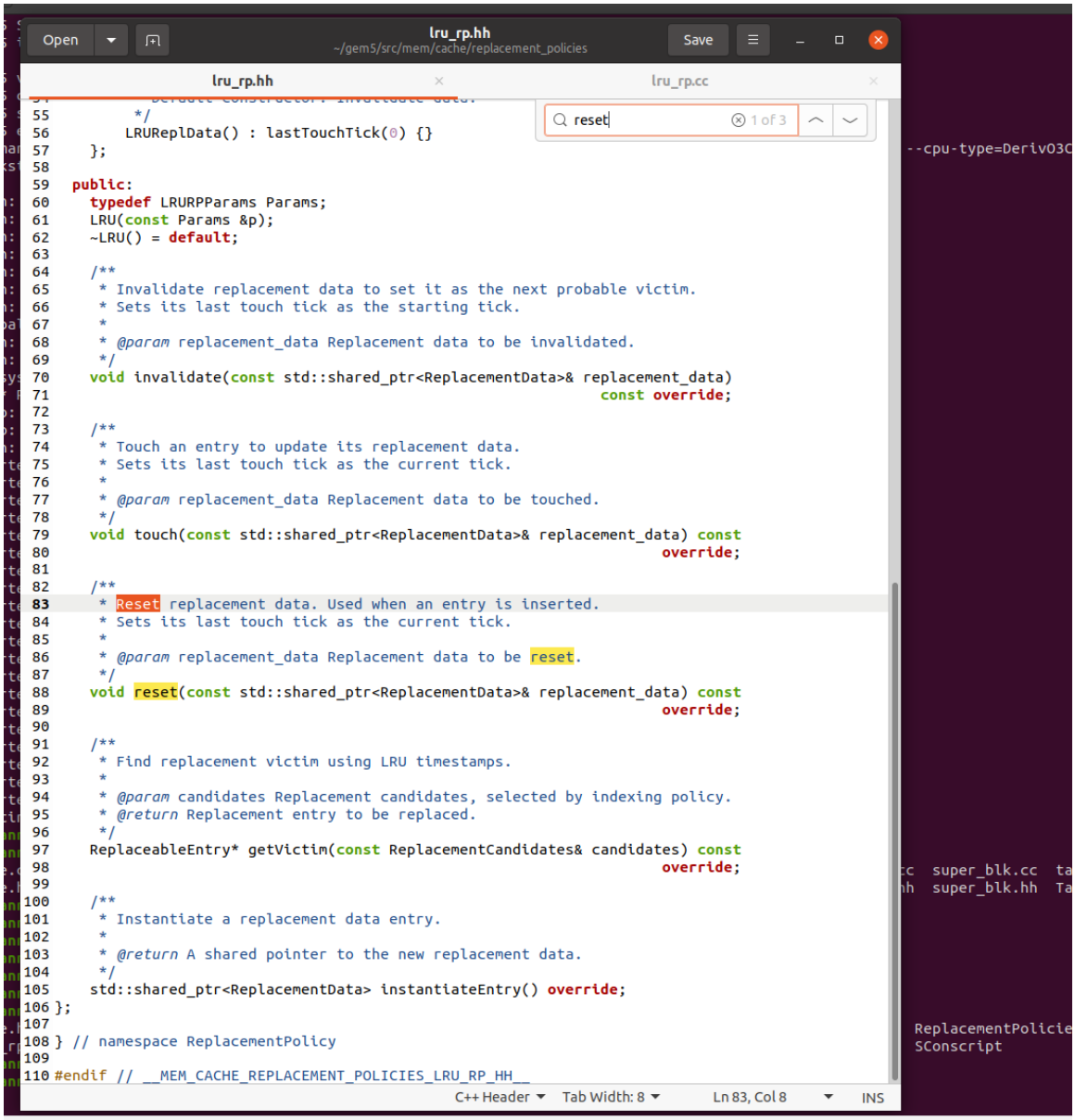
Purpose in Cache Management:
Ensures that newly inserted blocks start as MRU, so they are not immediately chosen for eviction.

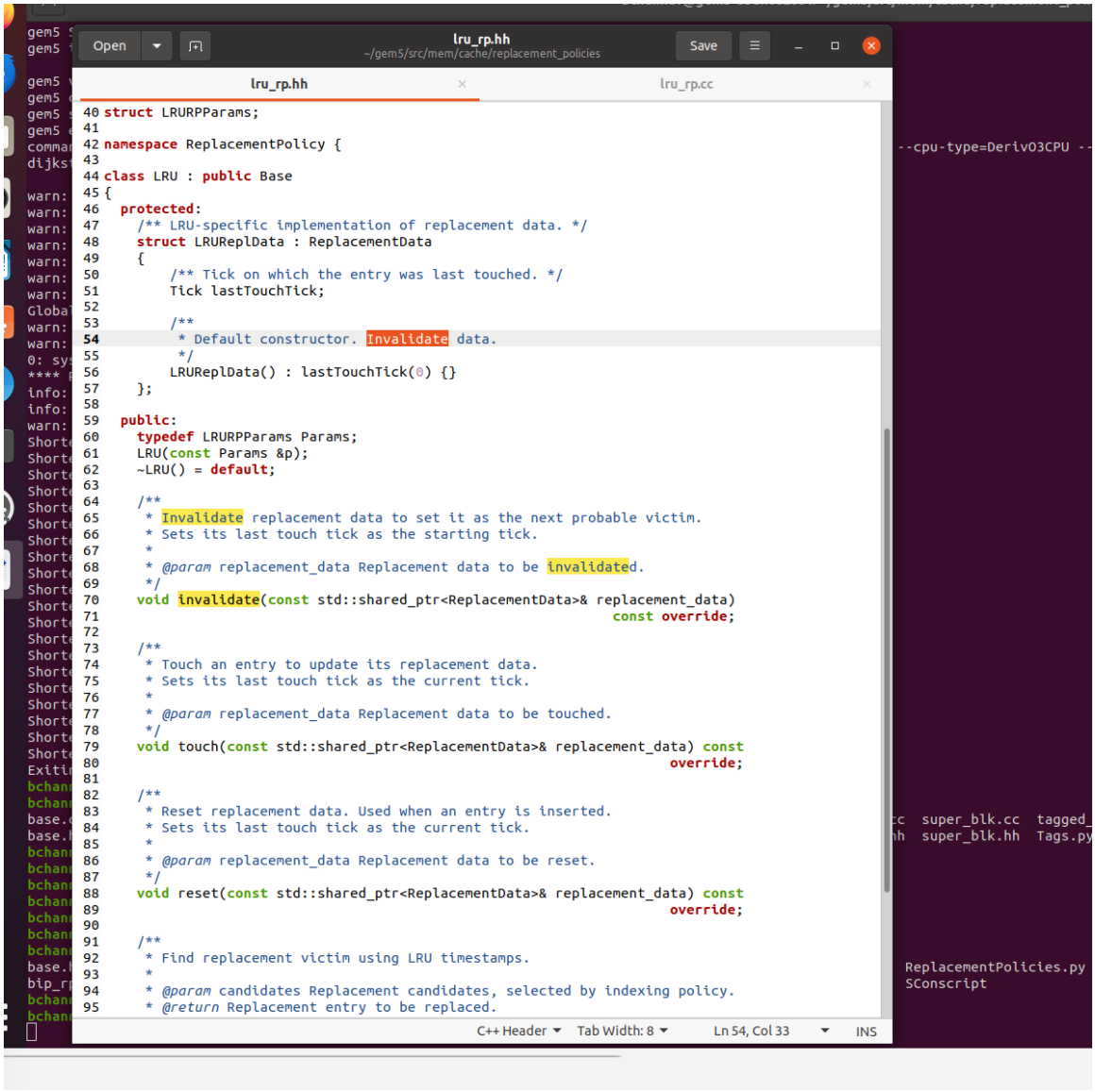
4. getVictim() – Choose a Block for Eviction

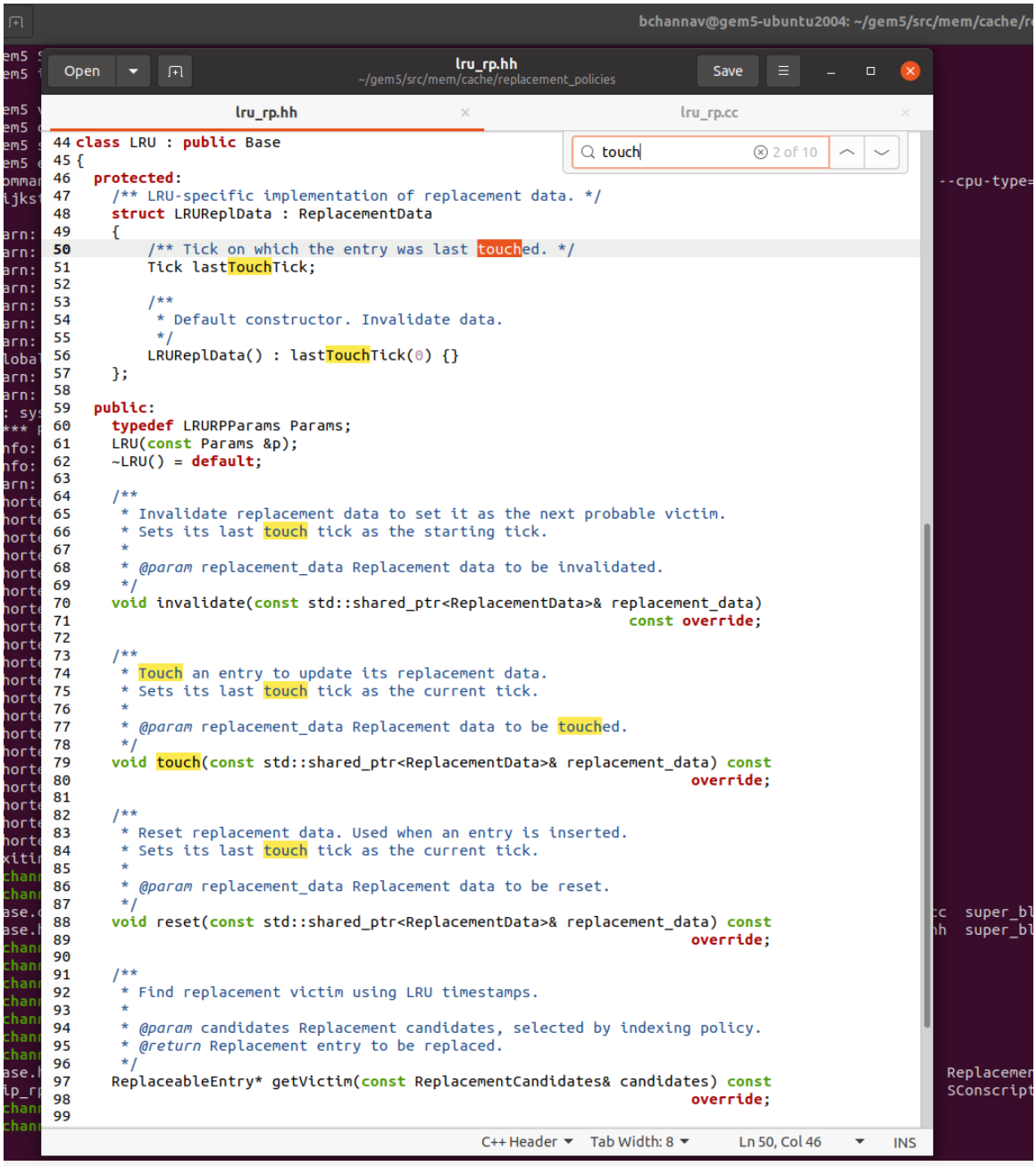


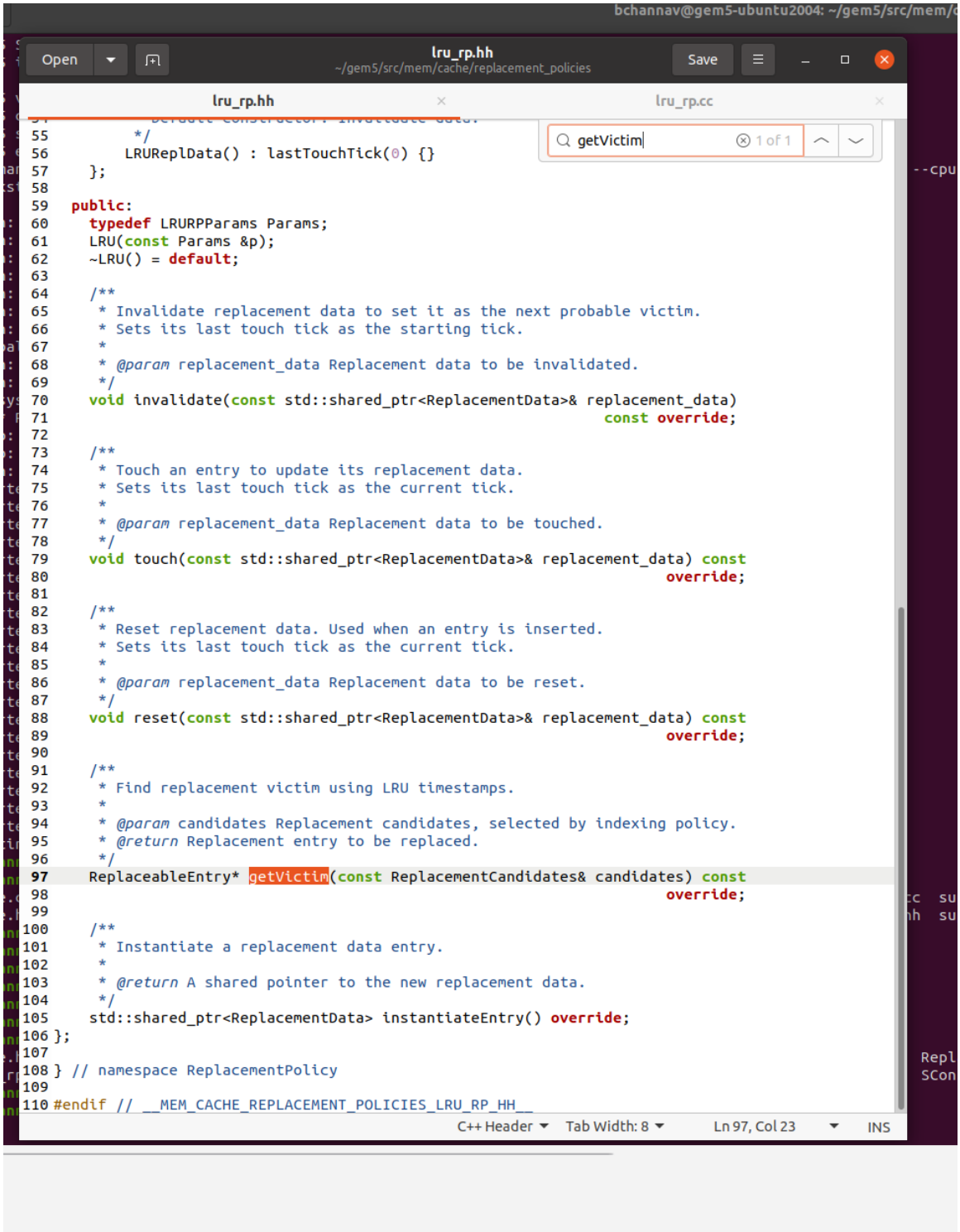
Description:
- Triggered when the cache is full and a new line must be inserted.
- Iterates through all candidate blocks in the set.
- Compares their lastTouchTick values and selects the block with the smallest timestamp.

Purpose in Cache Management:
Implements the LRU principle directly evict the block that has not been used for the longest time.


Open
lru_rp.hh
~/gem5/src/mem/cache/replacement_policies
Save
lru_rp.hh
lru_rp.cc
reset
1 of 3
*/
LRUReplData() : lastTouchTick(0) {}
};
public:
typedef LRURPParams Params;
LRU(const Params &p);
~LRU() = default;
/**
* Invalidate replacement data to set it as the next probable victim.
* Sets its last touch tick as the starting tick.
*
* @param replacement_data Replacement data to be invalidated.
*/
void invalidate(const std::shared_ptr<ReplacementData>& replacement_data)
const override;
/**
* Touch an entry to update its replacement data.
* Sets its last touch tick as the current tick.
*
* @param replacement_data Replacement data to be touched.
*/
void touch(const std::shared_ptr<ReplacementData>& replacement_data) const
override;
/**
* Reset replacement data. Used when an entry is inserted.
* Sets its last touch tick as the current tick.
*
* @param replacement_data Replacement data to be reset.
*/
void reset(const std::shared_ptr<ReplacementData>& replacement_data) const
override;
/**
* Find replacement victim using LRU timestamps.
*
* @param candidates Replacement candidates, selected by indexing policy.
* @return Replacement entry to be replaced.
*/
ReplaceableEntry* getVictim(const ReplacementCandidates& candidates) const
override;
/**
* Instantiate a replacement data entry.
*
* @return A shared pointer to the new replacement data.
*/
std::shared_ptr<ReplacementData> instantiateEntry() override;
};
} // namespace ReplacementPolicy
#endif // __MEM_CACHE_REPLACEMENT_POLICIES_LRU_RP_HH__
C++ Header
Tab Width: 8
Ln 83, Col 8
INS
--cpu-type=DerivO3C
super_blk.cc
super_blk.hh
ReplacementPolicie
SConscript


Open
lru_rp.hh
~/gem5/src/mem/cache/replacement_policies
Save
lru_rp.hh
lru_rp.cc
40 struct LRURPParams;
41
42 namespace ReplacementPolicy {
43
44 class LRU : public Base
45 {
46   protected:
47   /** LRU-specific implementation of replacement data. */
48   struct LRUReplData : ReplacementData
49   {
50   /** Tick on which the entry was last touched. */
51   Tick lastTouchTick;
52
53   /**
54   * Default constructor. Invalidate data.
55   */
56   LRUReplData() : lastTouchTick(0) {}
57   };
58
59   public:
60   typedef LRURPParams Params;
61   LRU(const Params &p);
62   ~LRU() = default;
63
64   /**
65   * Invalidate replacement data to set it as the next probable victim.
66   * Sets its last touch tick as the starting tick.
67   *
68   * @param replacement_data Replacement data to be invalidated.
69   */
70   void invalidate(const std::shared_ptr<ReplacementData>& replacement_data)
71   const override;
72
73   /**
74   * Touch an entry to update its replacement data.
75   * Sets its last touch tick as the current tick.
76   *
77   * @param replacement_data Replacement data to be touched.
78   */
79   void touch(const std::shared_ptr<ReplacementData>& replacement_data) const
80   override;
81
82   /**
83   * Reset replacement data. Used when an entry is inserted.
84   * Sets its last touch tick as the current tick.
85   *
86   * @param replacement_data Replacement data to be reset.
87   */
88   void reset(const std::shared_ptr<ReplacementData>& replacement_data) const
89   override;
90
91   /**
92   * Find replacement victim using LRU timestamps.
93   *
94   * @param candidates Replacement candidates, selected by indexing policy.
95   * @return Replacement entry to be replaced.
C++ Header
Tab Width: 8
Ln 54, Col 33
INS
--cpu-type=DerivO3CPU --
super_blk.cc tagged_
super_blk.hh Tags.py
ReplacementPolicies.py
SConscript


bchannav@gem5-ubuntu2004: ~/gem5/src/mem/cache/r
Open
lru_rp.hh
~/gem5/src/mem/cache/replacement_policies
Save
lru_rp.hh
lru_rp.cc
touch
2 of 10
44 class LRU : public Base
45 {
46   protected:
47   /** LRU-specific implementation of replacement data. */
48   struct LRUReplData : ReplacementData
49   {
50   /** Tick on which the entry was last touched. */
51   Tick lastTouchTick;
52
53   /**
54   * Default constructor. Invalidate data.
55   */
56   LRUReplData() : lastTouchTick(0) {}
57   };
58
59   public:
60   typedef LRURPParams Params;
61   LRU(const Params &p);
62   ~LRU() = default;
63
64   /**
65   * Invalidate replacement data to set it as the next probable victim.
66   * Sets its last touch tick as the starting tick.
67   *
68   * @param replacement_data Replacement data to be invalidated.
69   */
70   void invalidate(const std::shared_ptr<ReplacementData>& replacement_data)
71   const override;
72
73   /**
74   * Touch an entry to update its replacement data.
75   * Sets its last touch tick as the current tick.
76   *
77   * @param replacement_data Replacement data to be touched.
78   */
79   void touch(const std::shared_ptr<ReplacementData>& replacement_data) const
80   override;
81
82   /**
83   * Reset replacement data. Used when an entry is inserted.
84   * Sets its last touch tick as the current tick.
85   *
86   * @param replacement_data Replacement data to be reset.
87   */
88   void reset(const std::shared_ptr<ReplacementData>& replacement_data) const
89   override;
90
91   /**
92   * Find replacement victim using LRU timestamps.
93   *
94   * @param candidates Replacement candidates, selected by indexing policy.
95   * @return Replacement entry to be replaced.
96   */
97   ReplaceableEntry* getVictim(const ReplacementCandidates& candidates) const
98   override;
99
C++ Header
Tab Width: 8
Ln 50, Col 46
INS



Final Answer (for Q2b)

Each cache line in the LRU replacement policy (lru_rp.hh and lru_rp.cc) maintains a timestamp called lastTouchTick that indicates when it was last utilized. When a block is invalidated, the invalidate() method sets this timestamp to 0. As a result, it is the oldest block and a suitable replacement. Every time the block is accessed, the touch() function adjusts the timestamp to reflect the most recent usage. Reset() treats a newly added block as recently used by providing it with a starting timestamp. The getVictim() function eliminates the block with the least lastTouchTick after examining the timestamps of every block in a set. The

'least recently used' rule is followed here. Together, these functions monitor how recently the cache set has been used.

3.[0.5pt] Answer the following question based on your understanding of the paper "Insertion and Promotion for Tree-Based PseudoLRU Last-Level Caches" and your answers to questions 1 and 2 above.

Question: Sketch out your plans to implement the LRU-IPV (Least Recently Used-Insertion/Promotion Vector). Describe the functions you will implement and the data structures/variables you will use. Give as much implementation detail as possible when describing your functions, as your answer will be graded on the correctness of your description.

Question 3 – LRU-IPV Implementation Plan (Aligned with Jiménez 2013)

The basic LRU approach is expanded by the LRU-IPV (Least Recently Used – Insertion/Promotion Vector) replacement policy, which incorporates a data-driven insertion and promotion mechanism first presented by Jiménez et al. (2013) in *Insertion and Promotion for Tree-Based PseudoLRU Last-Level Caches*. This updated design enhances cache performance while maintaining a low enough weight for Gem5 integration by providing flexible control over block promotion and insertion locations.

1. Conceptual Framework

Cache blocks are always inserted and promoted to the MRU (most recently used) position using conventional LRU. However, Jiménez et al. (2013) showed that this wastes capacity because of *dead* or one-time-use blocks (§2.2–2.5). They introduced the Insertion/Promotion Vector (IPV), in which the insertion position is defined by V[k] and the new location following a hit is defined by V[0..k] for a k-way set-associative cache. Millions of different configurations are made possible by this parameterization, which can then be optimized using genetic or heuristic search (§4). The evolved optimal vector [0 0 1 0 3 0 1 2 1 0 5 1 0 0 1 11 13] outperformed pure LRU (§2.6) by about 3% for a 16-way cache.

2. Metadata Architecture

Following gem5's metadata model, each cache set maintains a shared structure that tracks the recency of every block, while each block stores minimal per-entry data.

• **Per-Set Metadata:**
```cpp
struct SetMeta {
  std::vector<int> recency;   // Recency positions 0 (MRU) – W−1 (LRU)
  std::vector<int> wayAtPos;  // Inverse map (position → way)
};
```
• **Per-Block Metadata:**
```cpp
struct LRUIPVReplData : ReplacementData {
  std::shared_ptr<SetMeta> setMeta;
  int wayIdxInSet;
  bool valid;
};
```

Initialization assigns recency[i]=i ensuring every block begins with a unique ordering (0–W−1).

3. Function Design and Implementation

Based on BaseReplacementPolicy, the LRUIPV class (files *lruipv_rp.hh/.cc*) rewrites the fundamental methods to employ vector-based reasoning.

The function **instantiateEntry:** generates shared SetMeta for each set and affixes it to every block. restores the order of recency.

reset(): When a miss and insertion occurs, the insertion index V[k] is retrieved, clamped to [0, W−1], and the frequency of other entries ≥ V[k] is increased. This puts the new block in a unique position (for example, 13 instead of 0).

On a hit, **touch():** shifts the block from oldPos to newPos = V[oldPos](§2.3), relocating intermediate blocks to preserve ordering. For instance, oldPos = 10−newPos = 5 from V[10] = 5.

invalidate(): Removes the block from the active stack by setting the values of marks_recency = W and valid = false.

getVictim(): iterates through the candidates and chooses the item that matches LRU semantics under IPV control and has the maximum recency (W−1).

4. Algorithmic and Computational Analysis

Every update runs in O(W), which is acceptable for typical associativities ≥≥16. All recency values form a unique permutation to avoid duplication. The approach maintains the formality of the recency stack definition (§2.1.2) while allowing for data-driven flexibility. Custom vectors can replicate LRU (V[i]=0) or simulate policies such as Dynamic Insertion Policy and DRRIP

5. Validation and Evaluation Plan

Testing is done according to the methodology of Jimenez et al. (§4). Various vectors will be assessed on SPEC workloads and compared to LRU. Functional validation uses debug prints to show the recency vectors after every update. Important metrics include system.l2.overall_miss_rate::total and system.cpu.cache.overall_miss_rate::total. A slight (≥1 %) deviation from the baseline is acceptable according to the project criteria.

6. Future Innovation and AI Integration

Section 4.2 of the paper explains how genetic algorithms find optimal IPVs offline. Our design expands on this idea to allow for automated training of the vector using machine learning frameworks. Trained IPVs can be loaded at runtime (--ipv_config) to permit self-optimizing cache behavior, which places LRU IPV as a basis for future AI-driven cache management policies.

Final Summary

The LRU-IPV implementation plan brings Jiménez et al.’s (2013) Insertion/Promotion Vector concept into gem5’s LRU framework. It achieves low overhead, high tunability, and readiness for AI extensions through adaptive insertion and promotion vectors. By enhancing LRU with p rogrammable recency logic, this design provides fine-grained control over cache behavior and sets a modern precedent for intelligent microarchitectural design.