Understanding the Alpha-Beta Cache Code

A compact set-associative transposition table with CRC-protected entries

Original code

```
typedef struct
{
  ui64_t ABCS_key;
  union
  {
    ui64_t ABCS_data;
    struct
    {
      score_t ABCS_score;
      i8_t ABCS_depth;
      ui8_t ABCS_flags;
      i8_t ABCS_moves[NABCS_MOVES];
    };
  };
#ifdef DEBUG
  ui64_t ABCS_debug;
#endif
} alpha_beta_cache_slot_t;

local
  int probe_alpha_beta_cache(search_t *arg_with_search,
                             alpha_beta_cache_entry_t *arg_alpha_beta_cache,
                             alpha_beta_cache_slot_t *arg_alpha_beta_cache_slot,
                             int arg_erase)
{
  BEGIN_BLOCK(__func__)

  arg_with_search->S_total_alpha_beta_cache_nprobes++;

  int result = FALSE;

  alpha_beta_cache_entry_t *with_alpha_beta_cache_entry =
    arg_alpha_beta_cache +
    FAST_MODULO64(arg_with_search->S_board.B_key, nalpha_beta_cache_entries);

  for (int islot = 0; islot < NABCS_SLOTS; islot++)
  {
    if ((with_alpha_beta_cache_entry->ABCE_slots[islot].ABCS_key ^
         arg_with_search->S_board.B_key) &
        (~0ULL << 32))
      continue;

    *arg_alpha_beta_cache_slot = with_alpha_beta_cache_entry->ABCE_slots[islot];

    ui32_t abcs_crc32 = (ui32_t)(arg_alpha_beta_cache_slot->ABCS_key ^
                                 arg_with_search->S_board.B_key);

    ui64_t abcs_data =
      (arg_alpha_beta_cache_slot->ABCS_data ^ arg_with_search->S_board.B_key);

    if (abcs2crc32(arg_with_search->S_board.B_key, abcs_data) == abcs_crc32)
    {
      arg_alpha_beta_cache_slot->ABCS_key ^= abcs_crc32;

      HARDBUG(arg_alpha_beta_cache_slot->ABCS_key !=
              arg_with_search->S_board.B_key)

      arg_alpha_beta_cache_slot->ABCS_data = abcs_data;

#ifdef DEBUG
      HARDBUG(with_alpha_beta_cache_entry->ABCE_slots[islot].ABCS_debug !=
              arg_with_search->S_board.B_key)
#endif

      result = TRUE;

      if (arg_erase)
        with_alpha_beta_cache_entry->ABCE_slots[islot] =
          alpha_beta_cache_slot_default;

      break;
    }
    else
    {
      arg_with_search->S_total_alpha_beta_cache_crc_errors++;
    }
  }

  END_BLOCK

  return (result);
}
```

```
local void
update_alpha_beta_cache(search_t *arg_with_search,
                        alpha_beta_cache_entry_t *arg_alpha_beta_cache,
                        alpha_beta_cache_slot_t *arg_alpha_beta_cache_slot)
{
  BEGIN_BLOCK(__func__)

  HARDBUG(arg_alpha_beta_cache_slot->ABCS_key != arg_with_search->S_board.B_key)

  alpha_beta_cache_entry_t *with_alpha_beta_cache_entry =
    arg_alpha_beta_cache +
    FAST_MODULO64(arg_with_search->S_board.B_key, nalpha_beta_cache_entries);

  int islot = INVALID;

  for (islot = 0; islot < NABCS_SLOTS; islot++)
  {
    if ((with_alpha_beta_cache_entry->ABCE_slots[islot].ABCS_key ^
         arg_with_search->S_board.B_key) &
        (~0ULL << 32))
      continue;

    alpha_beta_cache_slot_t abcs_slot =
      with_alpha_beta_cache_entry->ABCE_slots[islot];

    ui32_t abcs_crc32 =
      (ui32_t)(abcs_slot.ABCS_key ^ arg_with_search->S_board.B_key);

    ui64_t abcs_data = (abcs_slot.ABCS_data ^ arg_with_search->S_board.B_key);

    if (abcs2crc32(arg_with_search->S_board.B_key, abcs_data) == abcs_crc32)
    {
      abcs_slot.ABCS_key ^= abcs_crc32;

      HARDBUG(abcs_slot.ABCS_key != arg_with_search->S_board.B_key)

#ifdef DEBUG
      HARDBUG(with_alpha_beta_cache_entry->ABCE_slots[islot].ABCS_debug !=
              arg_with_search->S_board.B_key)
#endif

      break;
    }
  }

  if (islot == NABCS_SLOTS)
    islot = NABCS_SLOTS - 1;

  while (islot > 0)
  {
    with_alpha_beta_cache_entry->ABCE_slots[islot] =
      with_alpha_beta_cache_entry->ABCE_slots[islot - 1];

    --islot;
  }

  ui64_t abcs_crc32 = abcs2crc32(arg_alpha_beta_cache_slot->ABCS_key,
                                 arg_alpha_beta_cache_slot->ABCS_data);

  arg_alpha_beta_cache_slot->ABCS_key ^= abcs_crc32;
  arg_alpha_beta_cache_slot->ABCS_data ^= arg_with_search->S_board.B_key;

  with_alpha_beta_cache_entry->ABCE_slots[0] = *arg_alpha_beta_cache_slot;

#ifdef DEBUG
  with_alpha_beta_cache_entry->ABCE_slots[0].ABCS_debug =
    arg_with_search->S_board.B_key;
#endif

  END_BLOCK
}
```

Overview

This code implements a set-associative alpha-beta transposition table. Each hash-table bucket contains `NABCS_SL0TS` slots. A slot stores a position key, a compact 64-bit search result, and optionally a full debug key. Entries are ordered from newest to oldest, so insertion and updating use a simple recency-based replacement policy.

The cache also uses a 32-bit checksum. The checksum validates that the position key and payload belong together and helps detect ordinary hash collisions, corrupted entries, and inconsistent reads.

1. Slot layout

The anonymous union gives two views of the same eight payload bytes:

```
union
{
  ui64_t ABCS_data;
  struct
  {
    score_t ABCS_score;
    i8_t ABCS_depth;
    ui8_t ABCS_flags;
    i8_t ABCS_moves[NABCS_MOVES];
  };
};
```

The structured view is convenient for the search code. It exposes the score, depth, flags, and stored moves. The integer view allows the complete payload to be copied, XOR-encoded, and checksummed as one 64-bit value.

This design assumes that the anonymous structure occupies exactly eight bytes, including any compiler padding. That assumption is important enough to verify with a compile-time assertion.

```
_Static_assert(
  sizeof(((alpha_beta_cache_slot_t *)0)->ABCS_data) == 8,
  "ABCS_data must be 64 bits");
```

2. How a slot is encoded

Let `K` be the 64-bit board key, `D` the 64-bit payload, and `C` the 32-bit checksum:

```
C = abcs2crc32(K, D)
stored_key  = K ^ C
stored_data = D ^ K
```

Because the checksum occupies only the low 32 bits, XORing it into the key leaves the upper 32 bits unchanged. Those upper bits therefore remain available as a cheap preliminary signature.

During a probe, the payload and checksum are recovered as follows:

```
C = (ui32_t)(stored_key ^ K)
D = stored_data ^ K
```

The entry is accepted only when `abcs2crc32(K, D)` equals the recovered checksum.

3. Probing the cache

The probe begins by incrementing a counter and selecting one bucket:

```
entry = table + FAST_MODULO64(board_key,
                              nalpha_beta_cache_entries);
```

All slots in that bucket are then examined from slot 0 upward. Since updated entries are moved to slot 0, recent entries are tested first.

Cheap upper-half test

```
if ((stored_key ^ board_key) & (~0ULL << 32))
  continue;
```

This rejects the slot when any of the upper 32 key bits differ. It is equivalent to comparing `stored_key >> 32` with `board_key >> 32`. Most unrelated positions are discarded here without computing a checksum.

Decode and validate

```
ui32_t crc = (ui32_t)(stored_key ^ board_key);
ui64_t data = stored_data ^ board_key;

if (abcs2crc32(board_key, data) == crc)
  /* valid hit */
```

If the checksum matches, the copied slot is converted back to its normal decoded representation. The key is restored by XORing the checksum again, and the decoded payload is assigned to `ABCS_data`. The caller therefore receives an ordinary slot whose score, depth, flags, and moves can be read directly.

Debug verification and erasure

In debug builds, `ABCS_debug` stores the original full board key. It provides an independent assertion that a checksum-validated hit really belongs to the requested position. When `arg_erase` is true, a successful probe also replaces the stored slot with the default empty slot.

CRC failures

A CRC failure means that the upper 32 key bits matched, but the complete key/payload combination did not validate. This can be an ordinary partial-key collision. In a lock-free or shared cache it may also indicate that the reader observed an entry while another thread was replacing it.

4. Updating the cache

The update function first asserts that the supplied slot is in decoded form and belongs to the current board position. It then selects the same bucket and searches for an existing valid entry for that position using the same upper-half and CRC checks as the probe.

Replacement policy

When the position already exists, its current slot index is used. When it is absent, the final slot is selected:

```
if (islot == NABCS_SLOTS)
  islot = NABCS_SLOTS - 1;
```

The entries before that slot are shifted one place toward the back, and the new entry is inserted at slot 0. For example:

```
Existing entry X in slot 2:
  before: A B X C
  after:  X A B C

New entry X:
  before: A B C D
  after:  X A B C
```

This resembles least-recently-used replacement, but it is not strict LRU: successful probes do not move an entry forward. The ordering records insertion or update recency, not read recency.

Encoding before storage

```
crc = abcs2crc32(slot->ABCS_key, slot->ABCS_data);
slot->ABCS_key  ^= crc;
slot->ABCS_data ^= board_key;
entry->ABCE_slots[0] = *slot;
```

The new slot is checksummed and XOR-encoded before being written into slot 0. In debug builds, the original board key is stored separately in `ABCS_debug`.

Important side effect

> The function modifies the caller-provided slot in place. After `update_alpha_beta_cache()` returns, its key and payload remain encoded. The caller must not continue treating that object as a decoded cache result. A less surprising interface would encode a local copy and leave the argument unchanged.

5. Small mathematical example

```
K = 0x11223344_AABBCCDD
D = 0x01020304_05060708
C = 0x00000000_89ABCDEF

stored_key = K ^ C
           = 0x11223344_22100132

stored_data = D ^ K
```

The upper half of `stored_key` is still `0x11223344`. On lookup, XORing the stored key with `K` recovers `C`, while XORing the stored data with `K` recovers `D`. The recomputed checksum must then equal `C`.

6. What the design achieves

- Set associativity: several positions may occupy the same hash bucket.
- Compact payload: score, depth, flags, and moves share one 64-bit word.
- Fast rejection: the upper 32 key bits reject most non-matching slots.
- CRC validation: the full key and decoded payload must form a valid pair.
- Recency-based replacement: new and updated entries move to the front.
- Debug verification: debug builds retain the complete original key.

7. Details worth checking

- Confirm that the anonymous payload structure is exactly eight bytes on every supported compiler and ABI.
- Document or remove the in-place encoding side effect of `update_alpha_beta_cache()`.
- If multiple threads access the cache concurrently, verify the alignment, atomicity, and memory-order assumptions. The CRC detects many inconsistent reads, but it does not by itself make data races valid C.
- The variable used for the checksum in the update function is declared as `ui64_t`, although the checksum is conceptually 32 bits. Declaring it as `ui32_t` would make that intent clearer.

Conclusion

The code is a compact transposition-table implementation designed for fast lookup and collision safety. It stores a 64-bit search payload together with a partial key and a recoverable CRC. The probe cheaply filters slots by the upper half of the key, decodes the candidate payload, and validates the complete key/data pair. The update routine maintains a most-recently-updated ordering within each bucket and replaces the oldest slot when necessary.