Simulator Modifications

Overview

This project extends the baseline SimpleScalar cache simulator to model several different cache mechanisms that are used to reduce cache miss rates and memory traffic. The original simulator models a conventional two-level cache hierarchy with configurable DL1 and UL2 caches. This work augments the DL1 miss path to support additional structures and statistics collection while preserving the original simulator behavior for baseline configurations.

The primary goal of these modifications is comparative evaluation, not cycle-accurate modeling of real hardware. All mechanisms are implemented within SimpleScalar's existing framework for fair comparisons.

Baseline Cache Behavior

In the unmodified simulator, the cache access path is as follows:

1. Instruction or data access probes DL1.
2. On DL1 hit, access completes
3. On DL1 miss:
   - Access is forwarded to UL2
   - UL2 hit resolves the request
   - UL2 miss results in a memory access
4. Miss statistics and access counters are updated accordingly.

This baseline behavior serves as the reference point for all normalized results.

Added Cache Mechanisms

The following cache mechanisms were added to the DL1 miss path:

- Victim Cache (VC)
- Miss Cache (MC)
- Stream Buffer (SB)
- Combined mechanisms (VC and SB, MC and SB)

Victim Cache (VC)

Conceptual Model

The victim cache is modeled as a small, fully associative buffer that stores recently evicted DL1 blocks. On a DL1 miss, the victim cache is probed before forwarding the request to UL2.

Implementation

- The VC is checked only on DL1 misses
- On a VC hit:
  - The block is swapped back into DL1

  - UL2 is not accessed
- On a VC miss:
  - Normal DL1 miss handling continues

The victim cache capacity is parameterized by the number of entires, `vc_entries`

Statistics Added

- VC accesses
- VC hits
- VC hit rate

Modeling Assumptions

- Fully associative lookup
- No additional timing penalty
- Perfect replacement Policy (LRU behavior)

Miss Cache (MC)

Conceptual Model

The miss cache captures recent DL1 miss addresses and detects short-term reuse that would otherwise go to UL2 repeatedly.

Unlike a victim cache, the miss cache does not store data, only tags.

Implementation Details

- On a DL1 miss:
  - Miss cache is probed
- On MC hit:
  - The miss is counted as resolved at the MC
  - UL2 access is bypassed
- On MC miss:
  - Address is inserted into the miss cache
  - Normal UL2 access proceeds

The miss cache capacity is parameterized by the number of entries, `mc_entries`

Statistics Added

- MC accesses
- MC hits
- MC hit rate

Modeling Assumptions

- Tag-only structure
- Fully associative
- No false positives

- No bandwidth or latency modeling

Stream Buffer (SB)

Conceptual Model

The stream buffer models a simple sequential prefetcher. When a DL1 miss occurs, a stream of future blocks is prefetched into the buffer.

Implementaiton Details

- Activated on DL1 misses
- Prefetches a stream of sequential addresses
- On subsequent DL1 misses:
  - Stream buffer is probed
- On SB hit:
  - Block is supplied from the buffer
  - UL2 access is avoided
- On SB miss:
  - A new stream may be allocated

The stream buffer depth is parameterized by `sb_depth`.

Statistics Added

- SB accesses
- SB hits
- SB hit rate

Modeling Assumptions

- Perfect stream detection
- Fixed stride of +1 cache block
- No prefetch pollution modeling
- No bandwidth contention

Combined Mechanisms

Two combined mechanisms were implemented:

Victim Cache + Stream Buffer

Access order on DL1 miss:

1. Victim Cache
2. Stream buffer
3. Ul2

Miss Cache + Stream Buffer

Access order on DL1 miss:

1. Miss cache
2. Stream buffer
3. UL2

Access Ordering and Priority

For all modified configurations, the following order is used:

1. DL1
2. Auxiliary structure/structures
3. UL2
4. Memory

Only one auxiliary structure may satisfy a miss. Once a hit occurs, downstream structures are not accessed.

Counter and Statistic Integration

All new counters were integrated using SimpleScalar's existing statistics framework. This ensures:

- Consistent accounting
- Comparable miss rates
- Minimal disruption to existing output formats

Derived metrics are computed in post-processing scripts outside of the simulator.

Design Constraints and Simplifications

The following simplifications were made:

- No cycle-accurate timing
- No modeling of contention or bandwidth
- No pollution effects between structures
- No dynamic adaptation or prefetch behavior
- Single-core, single-thread execution

These constraints keep the simulations focused on behavioral trends.

Simulator Modifications (`sim-cache.c`)

This part of the document describes specific modifications made to the sim-cache simulator for evaluation.

1. Added experiment controls and state

The simulator now includes experiment toggles, configuration parameters, and per-mechanism state, including simple fully-associative storage for VC and MC, and metadata for stream buffers.

```
// Experiment controls: victim cache, miss cache, and stream buffer

// Experiment mode (for stats/reporting)
static char *exp_mode = "baseline"; // baseline, victim, miss, stream,
stream_multi, victim_stream

// Victim cache for DL1
static int vc_enable = FALSE;
static int vc_entries = 4;

// Victim buffer state: fully-associative buffer of block addresses
static md_addr_t *vc_lines = NULL;
static int *vc_valid = NULL;

// Miss cache
static int mc_enable = FALSE;
static int mc_entries = 4;

// Miss cache (DL1) – fully-associative buffer of block addresses
static md_addr_t *mc_lines = NULL;
static int *mc_valid = NULL;

// Miss cache statistics
static counter_t mc_lookups = 0; // number of times we probed the miss cache
static counter_t mc_hits = 0;    // number of times miss cache hit

// Stream buffer statistics
static counter_t sb_lookups = 0;    // DL1 read misses checked in stream buffers
static counter_t sb_hits = 0;       // stream buffer hit count
static counter_t sb_prefetches = 0; // prefetches issued from stream buffers

// Stream buffer configuration
static int sb_enable = FALSE;
static int sb_count = 1;
static int sb_depth = 4;
static int sb_degree = 1;

// Stream buffer metadata structure
struct stream_buffer_t
{
  md_addr_t base_line_addr; // aligned address of current stream start
  int valid;
  int depth;
  int head;                 // index of next line to consume
  int filled;
};

static struct stream_buffer_t *sbuffers = NULL;
```

Notes:

- VC and MC are modeled as fully-associative address buffers (`mc_addr_t` block addresses), with simple validity bits.

- Stream buffers store metadata only (base line, head pointer, depth). Prefetching is modelled by issuing accesses into the UL2 path.

2. Modified DL1 miss handler for VC to MC to SB behavior

Significant changes are made to `dl1_access_fn()`, which now performs:

1. Victim cache probe on DL1 read miss (if enabled)
2. Miss cache probe on DL1 read miss (if enabled)
3. Stream buffer probe/allocation on DL1 read miss (if enabled)
4. If none hit, proceed to UL2 (original behavior)

```
dl1_access_fn(enum mem_cmd cmd,     // access cmd, Read or Write
          md_addr_t baddr,      // block address to access
          int bsize,        // size of block to access
          struct cache_blk_t *blk,  // ptr to block in upper level
          tick_t now)       // time of access
{
  static int mc_repl_index = 0; // round-robin for miss cache
  static int vc_repl_index = 0;  // round-robin for victim buffer
  unsigned int lat;

  // If victim cache is enabled, check victim buffer first on DL1 miss.
  // Check only for READs.
  if (vc_enable && vc_lines != NULL && vc_valid != NULL && cmd == Read)
    {
      int i;

      // look in victim cache after dl1 miss
      vc_lookups++;

      for (i = 0; i < vc_entries; i++)
        {
          if (vc_valid[i] && vc_lines[i] == baddr)
            {
              // Victim hit: satisfied by victim buffer, do not go to DL2
              vc_hits++;

              vc_valid[i] = 0; // consume this victim entry
              return 1;        // nominal latency
            }
        }
    }

  // Miss cache lookup (only on reads, after victim cache)
  if (mc_enable && mc_lines != NULL && mc_valid != NULL && cmd == Read)
    {
      int j;

      mc_lookups++;

      for (j = 0; j < mc_entries; j++)
```

```
        {
          if (mc_valid[j] && mc_lines[j] == baddr)
            {
              // Miss hit: satisfied by miss cache, do not go to DL2
              mc_hits++;
              return 1; // nominal latency
            }
        }
    }

  // No victim hit (or victim disabled): go to next level of cache hierarchy
  // Stream buffer: probe before going to DL2 (read misses only)
  if (sb_enable && sbuffers != NULL && cmd == Read)
    {
      md_addr_t line_addr = baddr & ~((md_addr_t)(bsize - 1)); // align to block size
      int hit_buf = -1;
      int hit_offset = -1;
      int i;

      sb_lookups++;

      // Search existing streams for this line
      for (i = 0; i < sb_count; i++)
        {
          if (!sbuffers[i].valid)
            continue;

          md_addr_t base = sbuffers[i].base_line_addr;

          if (line_addr < base)
            continue;

          md_addr_t diff = line_addr - base;

          // Must be aligned to block size
          if ((diff % (md_addr_t)bsize) != 0)
            continue;

          int offset = (int)(diff / (md_addr_t)bsize);

          // Only a hit if it is at or ahead of head, and wihthin FILLED
          if (offset < sbuffers[i].head || offset >= sbuffers[i].filled)
            continue;

          hit_buf = i;
          hit_offset = offset;
          break;
        }

      if (hit_buf >= 0)
        {
          // Stream buffer hit: satisfy from SB, no DL2 access
          sb_hits++;

          // Advance head to one past this block
          sbuffers[hit_buf].head = hit_offset + 1;

          // Invalidate stream if everything prefetched is consumed
          if (sbuffers[hit_buf].head >= sbuffers[hit_buf].filled)
            sbuffers[hit_buf].valid = FALSE;

          return 1; // nominal latency
        }
      else
        {
          // Miss in all stream buffers, allocate/refresh a buffer for new
stream
          static int sb_repl_index = 0;
          int idx = sb_repl_index;

          sb_repl_index++;
          if (sb_repl_index >= sb_count)
            sb_repl_index = 0;

          sbuffers[idx].base_line_addr = line_addr;
          sbuffers[idx].valid = TRUE;
          sbuffers[idx].head = 1;   // base line is being consumed now
          sbuffers[idx].filled = 1; // offset 0 is valid conceptually

          {
            int d;
            int max_depth = sbuffers[idx].depth;

            // Prefetch ALL remaining lines in this stream window: offsets
[1..depth-1]
            for (d = 1; d < max_depth; d++)
              {
                md_addr_t pf_addr = line_addr + (md_addr_t)d *
(md_addr_t)bsize;

                // Model prefetch into the hierarchy (UL2 if present;
otherwise memory).
                if (cache_dl2)
                  {
                    ul2_d_accesses++;
                    cache_access(cache_dl2, Read, pf_addr, NULL, bsize,
now,
                                 NULL, NULL);
                  }

                sb_prefetches++;
                sbuffers[idx].filled++; // now this offset is considered
present in SB
              }
          }
        }
```

```
    }
  if (cache_dl2)
    {
      // count DL1 to DL2 accesses
      ul2_d_accesses++;

      // access next level of data cache hierarchy
      lat = cache_access(cache_dl2, cmd, baddr, NULL, bsize,
                         /* now */now, /* pudata */NULL, /* repl addr */NULL);
    }
  else
    {
      // access main memory, which is always done in the main simulator loop
      lat = /* access latency, ignored */1;
    }

  // On READs, install this line into miss cache and victim buffer
  if (cmd == Read)
    {
      // Miss cache install
      if (mc_enable && mc_lines != NULL && mc_valid != NULL)
        {
          mc_lines[mc_repl_index] = baddr;
          mc_valid[mc_repl_index] = 1;

          mc_repl_index++;
          if (mc_repl_index >= mc_entries)
            mc_repl_index = 0;
        }

      // Victim buffer install
      if (vc_enable && vc_lines != NULL && vc_valid != NULL)
        {
          vc_lines[vc_repl_index] = baddr;
          vc_valid[vc_repl_index] = 1;

          vc_repl_index++;
          if (vc_repl_index >= vc_entries)
            vc_repl_index = 0;
        }
    }

  return lat;
}
```

Design notes:

- The hit behavior for VC/MC/SB returns a nominal latency of 1 (functional model).
- ul2_d_accesses is incremented for: -demand misses that go to UL2
  - SB “prefetch” accesses issued into UL2

- Replacement policy is deliberately simple (round-robin) to keep focus on comparative mechanism behavior rather than policy tuning.

3. Added CLI options to control experiments

New simulator options were registered so the bash wrappers can control modes and parameters without recompiling.

```
/ Experiment options: victim cache, miss cache, stream buffer

  opt_reg_string(odb, "-exp:mode",
         "experiment mode (baseline, victim, miss, stream, stream_multi, victim_stream)",
         &exp_mode, "baseline",
         /* print */TRUE, /* format */NULL);

  opt_reg_int(odb, "-vc:enable",
         "enable victim cache on DL1 (0/1)",
         &vc_enable, FALSE, /* print */FALSE, NULL);

  opt_reg_int(odb, "-vc:entries",
         "number of entries in victim cache",
         &vc_entries, 4, /* print */FALSE, NULL);

  opt_reg_int(odb, "-mc:enable",
         "enable miss cache on DL1 (0/1) [stub]",
         &mc_enable, FALSE, /* print */FALSE, NULL);

  opt_reg_int(odb, "-mc:entries",
         "number of entries in miss cache [stub]",
         &mc_entries, 4, /* print */FALSE, NULL);

  opt_reg_int(odb, "-sb:enable",
         "enable stream buffers (0/1)",
         &sb_enable, FALSE, /* print */FALSE, NULL);

  opt_reg_int(odb, "-sb:count",
         "number of independent stream buffers",
         &sb_count, 1, /* print */FALSE, NULL);

  opt_reg_int(odb, "-sb:depth",
         "depth (lines) per stream buffer",
         &sb_depth, 4, /* print */FALSE, NULL);

  opt_reg_int(odb, "-sb:degree",
         "prefetch degree per trigger",
         &sb_degree, 1, /* print */FALSE, NULL);
```

4. Added allocation/initialization for VC, MC, SB

During option checking, the simulator allocates per-mechanism state only when enabled.

```
// Extra structures: victim cache and miss cache

  // Victim buffer for DL1: small fully-associative buffer of block addresses
  if (vc_enable && cache_dl1 != NULL)
    {
      int i;

      vc_lines = (md_addr_t *)calloc(vc_entries, sizeof(md_addr_t));
      vc_valid = (int *)calloc(vc_entries, sizeof(int));
      if (!vc_lines || !vc_valid)
    fatal("out of memory for victim cache buffer");

      for (i = 0; i < vc_entries; i++)
    vc_valid[i] = 0;
    }

  // Miss cache: small fully-associative buffer of block addresses
  if (mc_enable && cache_dl1 != NULL)
    {
      int i;

      mc_lines = (md_addr_t *)calloc(mc_entries, sizeof(md_addr_t));
      mc_valid = (int *)calloc(mc_entries, sizeof(int));
      if (!mc_lines || !mc_valid)
    fatal("out of memory for miss cache buffer");

      for (i = 0; i < mc_entries; i++)
    mc_valid[i] = 0;
    }

  // Stream buffers: allocate metadata only
  if (sb_enable && sb_count > 0 && sb_depth > 0)
    {
      int i;

      sbuffers = (struct stream_buffer_t *)calloc(sb_count, sizeof(struct stream_buffer_t));
      if (!sbuffers)
    fatal("out of memory for stream buffers");

      for (i = 0; i < sb_count; i++)
    {
      sbuffers[i].valid = FALSE;
      sbuffers[i].depth = sb_depth;
      sbuffers[i].head  = 0;
      sbuffers[i].base_line_addr = 0;
      sbuffers[i].filled = 0;
    }
    }
```

5. Added statistics counters for evaluation

New counters are registered to allow hit-rate computations and UL2-demand tracking from the standard simulator output.

```
// victim cache stats
  stat_reg_counter(sdb, "vc_lookups",
                   "victim cache lookup count (DL1 read misses)",
                   &vc_lookups, 0, NULL);
  stat_reg_counter(sdb, "vc_hits",
                   "victim cache hit count",
                   &vc_hits, 0, NULL);
  // UL2 access statistics
  stat_reg_counter(sdb, "ul2_d_accesses",
                   "DL2 data-side accesses from DL1 miss handler",
                   &ul2_d_accesses, 0, NULL);
  stat_reg_counter(sdb, "ul2_i_accesses",
                   "IL2 instruction-side accesses from IL1 miss handler",
                   &ul2_i_accesses, 0, NULL);
  // miss cache statistics
  stat_reg_counter(sdb, "mc_lookups",
                   "miss cache lookup count (DL1 read misses after victim)",
                   &mc_lookups, 0, NULL);
  stat_reg_counter(sdb, "mc_hits",
                   "miss cache hit count",
                   &mc_hits, 0, NULL);
  // stream buffer statistics
  stat_reg_counter(sdb, "sb_lookups",
                   "stream buffer lookup count (DL1 read misses)",
                   &sb_lookups, 0, NULL);
  stat_reg_counter(sdb, "sb_hits",
                   "stream buffer hit count",
                   &sb_hits, 0, NULL);
  stat_reg_counter(sdb, "sb_prefetches",
                   "stream buffer prefetches issued (to UL2/memory)",
                   &sb_prefetches, 0, NULL);
```

Summary

The modified simulator extends SimpleScalar with modular, parameterized cache mechanisms that integrate directly into the DL1 miss path. While simplified, these models capture the main effects of victim caching, miss caching, and stream buffering, enabling controlled comparison across cache sizes, workloads, and mechanism combinations.