Two Level Cache Simulator Report

Project 2 Report

Abdallah Ibrahim, Aly Youssef, Mahmoud Aly

July 22, 2025

1. Introduction

As processors continue to outpace main memory in raw speed, the bottleneck has become the memory access time limiting overall system performance. To bridge this gap, modern CPUs employ multi-level cache hierarchies that transparently keep data as close to the CPU as possible. In this project, we've built a two-level, set-associative, write-back cache simulator with variable L1 line sizes. By driving the simulator with a variety of memory-access generators and measuring the resulting effective CPI, we explore how different L1 line sizes influence performance across diverse workloads and use these insights to identify cache configurations that give the best trade-offs under practical conditions.

2. Design and Implementation

2.1 Cache Configuration

- **L1 Cache**
  - Size: 16 KB
  - Line size: 16 B, 32 B, 64 B, 128 B (variable)
  - Associativity: 4-ways
  - Hit time: 1 cycle
- **L2 Cache**
  - Size: 128 KB
  - Line size: 64 B (fixed)
  - Associativity: 8-ways
  - Hit time: 10 cycles
- **Main Memory (DRAM)**
  - Size: 64 GB
  - Access penalty: 50 cycles

2.2 Assumptions

- 35% of the instructions are loads and stores (memory operations), and 50% of these memory accesses are reads.
- Instructions are fetched from an ideal memory and do not affect CPI.

- Caches employ a write-back policy.
- Random replacement is used.
- The CPU CPI is 1 under an ideal caching system (100% hit rate in L1 cache).

2.3 Caching Algorithm

The simulator executes a main loop of 1,000,000 instructions. On each iteration:

1. Generate a uniform random number $r_1 \in [0, 1]$. If $r_1 \leq 0.35$, perform a memory operation; otherwise, proceed to the next instruction.
2. Generate a second random number $r_2 \in [0, 1]$ to choose the operation type:
   - If $r_2 \leq 0.5$, it is a *store*.
   - Otherwise, it is a *load*.
3. Generate an address using the selected memory generator, then send the address and the operation (read/write) to the L1 cache.
4. **Access L1:**
   - Increment the cycle counter by the L1 hit time (1 cycle).
   - If the access is a hit, mark the line as dirty if it was a store, then continue to the next instruction.
   - If it is a miss, proceed to access the L2 cache.
5. **Access L2:**
   - Increment the cycle counter by the L2 hit time (10 cycles).
   - If the access is a hit, mark the line as dirty on a store, then continue.
   - If it is a miss, proceed to access main memory (DRAM).
6. **Access DRAM:**
   - Increment the cycle counter by the DRAM penalty (50 cycles).
7. **Line refill and replacement:**
   - On an L1 miss (after potentially hitting or missing in L2/DRAM), bring the requested line into L1.
   - On an L2 miss, also bring the line into L2 before inserting into L1.
   - Use random replacement: if there is an invalid line, use it; otherwise evict a random line.
   - If the evicted line is dirty, write it back (taking into account the cycle penalty in that cache level).
   - Insert the new line and, if the original operation was a store, mark it dirty.

2.4 Cache Implementation

The cache hierarchy is realized by a reusable `Cache` class that can be chained to form multiple levels. Only the constructor and usage are shown here; the full implementation resides in `cache.cpp`.

Constructor Signature

```
Cache(size_t cacheSizeBytes,
      size_t blockSizeBytes,
      size_t associativity,
      unsigned hitTimeCycles,
      unsigned memTimeCycles,
      Cache *nextLevel = nullptr);
```

- `cacheSizeBytes` – total capacity of this cache (bytes).
- `blockSizeBytes` – size of each cache line (bytes).
- `associativity` – number of lines per set (e.g. 4 for 4-way associative).
- `hitTimeCycles` – cycle penalty for a hit in this level.
- `memTimeCycles` – cycle penalty on a miss when there is no next level.
- `nextLevel` – pointer to the next cache level (or `nullptr`).

Instantiating L2 and L1

```
Cache L2(128 * 1024,  // 128KB
         64,          // 64B line
         8,           // 8-way
         10,          // 10 cycles hit time
         50,          // 50 cycles to DRAM on L2 miss
         nullptr);    // no further level

Cache L1(16 * 1024,   // 16KB
         32,          // 32B line (variable)
         4,           // 4-way
         1,           // 1 cycle hit time
         0,           // miss penalty handled by L2
         &L2);        // next level = L2
```

High-Level Behavior

- An `access(addr, op)` call first probes the current cache:
  - On a hit, returns `hitTimeCycles` (and marks the line dirty if `STORE`).
  - On a miss, recursively calls `nextLevel->access(...)` or applies `memTimeCycles`, then installs the line here.
- Write-back policy: evicted dirty lines cause a **store** to the next level (or main memory).
- This design is easily extended to three or more levels by creating additional `Cache` objects and linking them via the `nextLevel` pointer.

3. Results

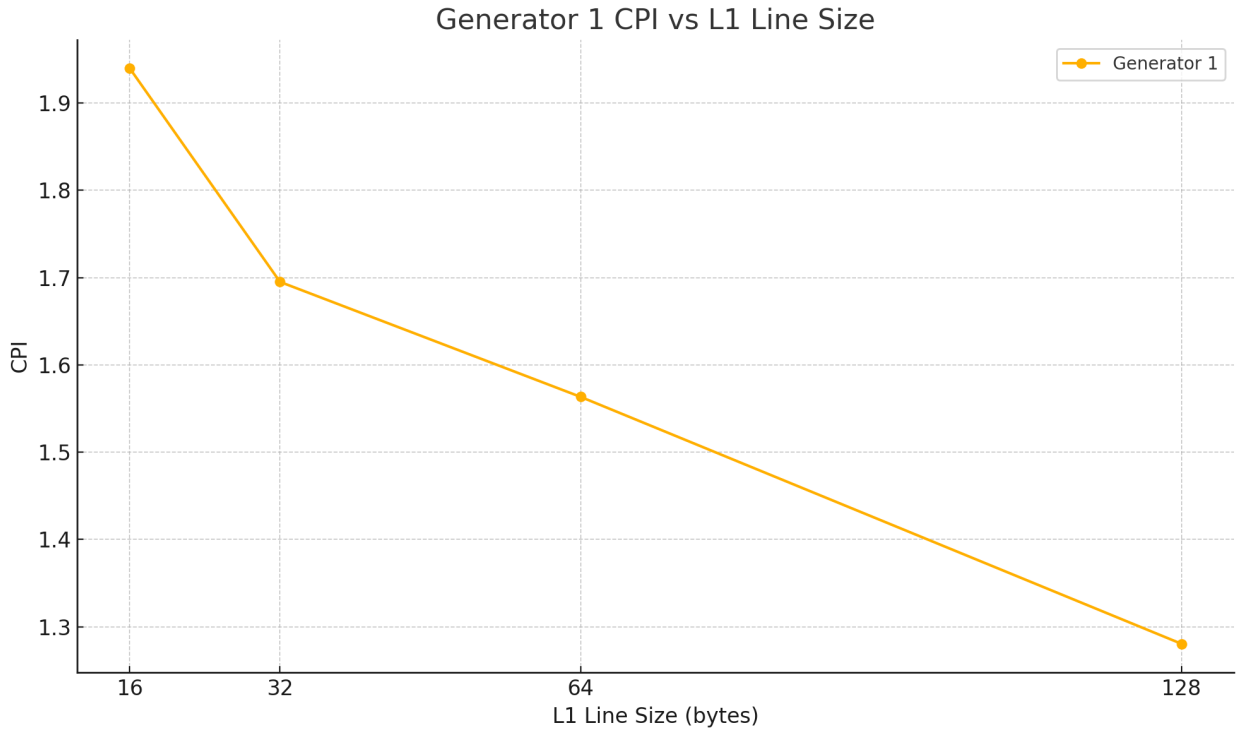
3.1 Experimental Configuration

To evaluate cache performance, we combined five distinct memory-access generators with four different L1 cache line sizes:

- **Memory generators:** 5 (random, sequential)
- **L1 line sizes tested:** 16 B, 32 B, 64 B, 128 B

Each of the 5x4 = 20 configurations was simulated for 1,000,000 instructions, holding all other cache and memory parameters constant.

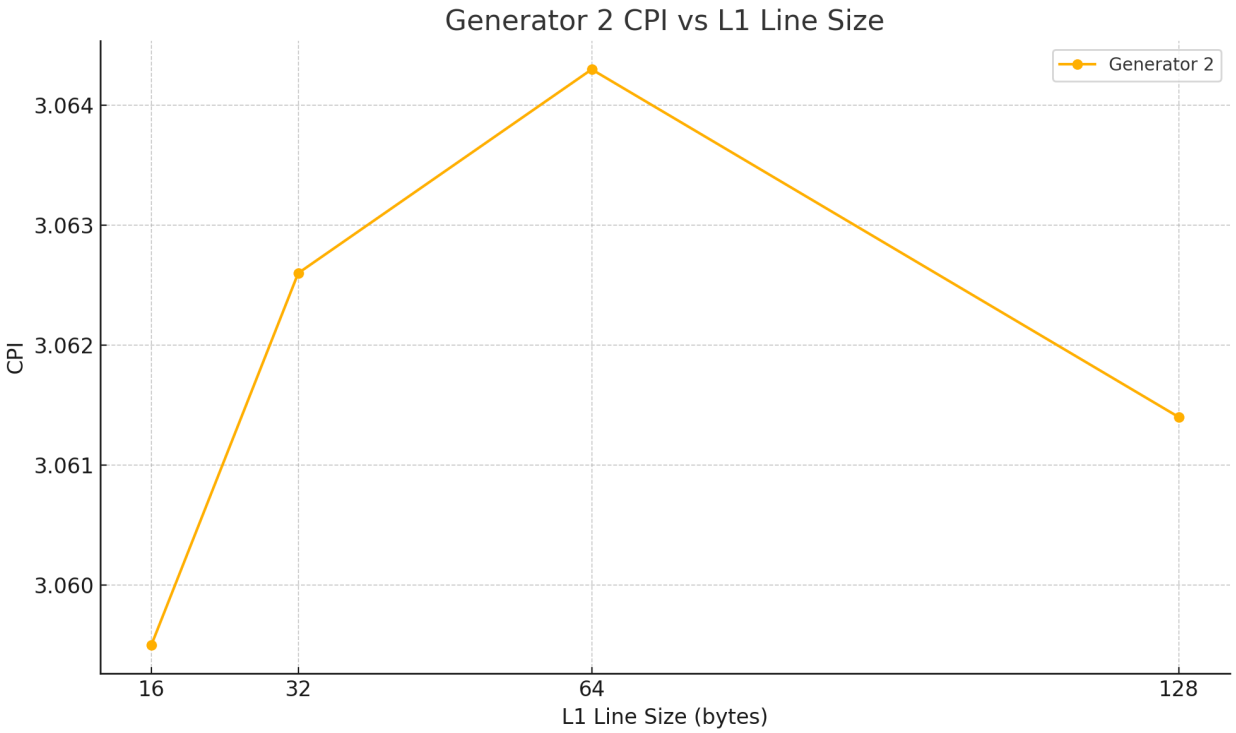
3.2 Generator 1



Line Size (B)	CPI	L1 Hit Rate	L1 Miss Rate	L2 Hit Rate	L2 Miss Rate
16	1.9395	0.9375	0.0625	0.8547	0.1453
32	1.6950	0.9687	0.0313	0.7232	0.2768
64	1.5634	0.9844	0.0156	0.4611	0.5389
128	1.2806	0.9922	0.0078	0.4659	0.5341

Sequentially stepping through all 64GB gives very strong spatial locality, where larger L1 lines capture more contiguous addresses, so the L1 hit rate increasing from about 94% at 16B up to nearly 100% at 128B, driving CPI down accordingly.

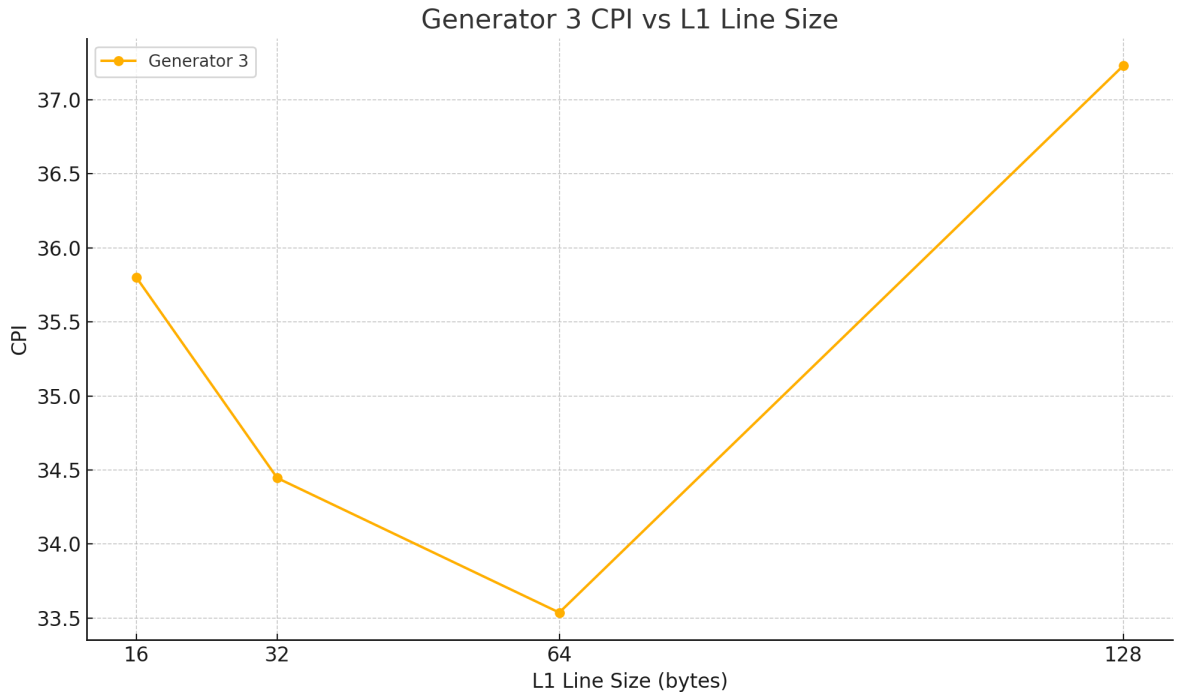
3.3 Generator 2



Line Size (B)	CPI	L1 Hit Rate	L1 Miss Rate	L2 Hit Rate	L2 Miss Rate
16	3.0595	0.6653	0.3347	0.9981	0.0019
32	3.0626	0.6653	0.3347	0.9981	0.0019
64	3.0643	0.6656	0.3344	0.9981	0.0019
128	3.0614	0.6663	0.3337	0.9981	0.0019

With random address generation in a 24KB range that is larger than the 16KB L1 but way under the 128KB L2, L1 hit rates stay low around 66% independent of line size, since 24KB won't fit in L1. However, almost every L1 miss is hit by L2, so L2 hit rate is near 100% keeping CPI almost constant across line sizes.

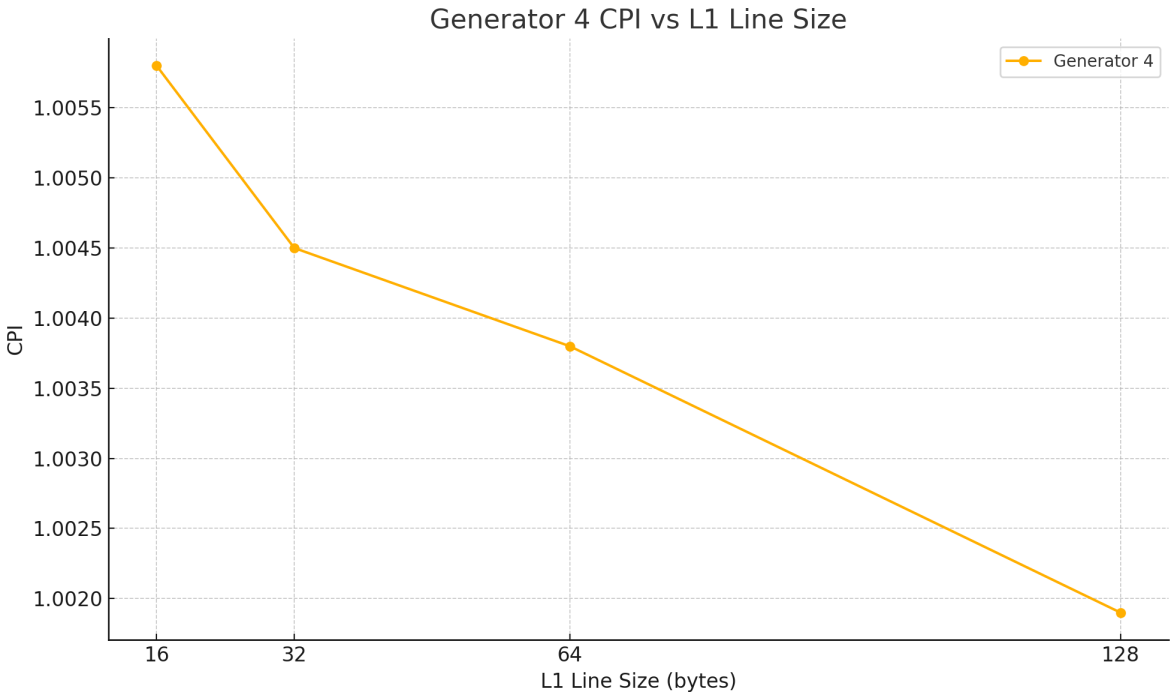
3.4 Generator 3



Line Size (B)	CPI	L1 Hit Rate	L1 Miss Rate	L1 WB Count	L2 Hit Rate	L2 Miss Rate	L2 WB Count
16	35.8017	0.0003	0.9997	175,320	0.2074	0.7926	174,296
32	34.4458	0.0003	0.9997	175,359	0.2591	0.7409	174,271
64	33.5372	0.0003	0.9997	175,516	0.2939	0.7061	174,402
128	37.2300	0.0002	0.9998	175,497	0.1536	0.8464	174,633

Uniform random accesses across the full 64GB give us zero locality, so L1 hit rates are basically 0% and L2 hit rates are low (20% - 30%) regardless of line size as every new address tends to miss both caches. An interesting observation is when the line size of L1 cache is larger than L2 (128 and 64), the CPI spikes. That means each L1 miss spans two L2 blocks instead of one. That doubles write-back traffic at the L2 level, increasing the CPI.

3.5 Generator 4

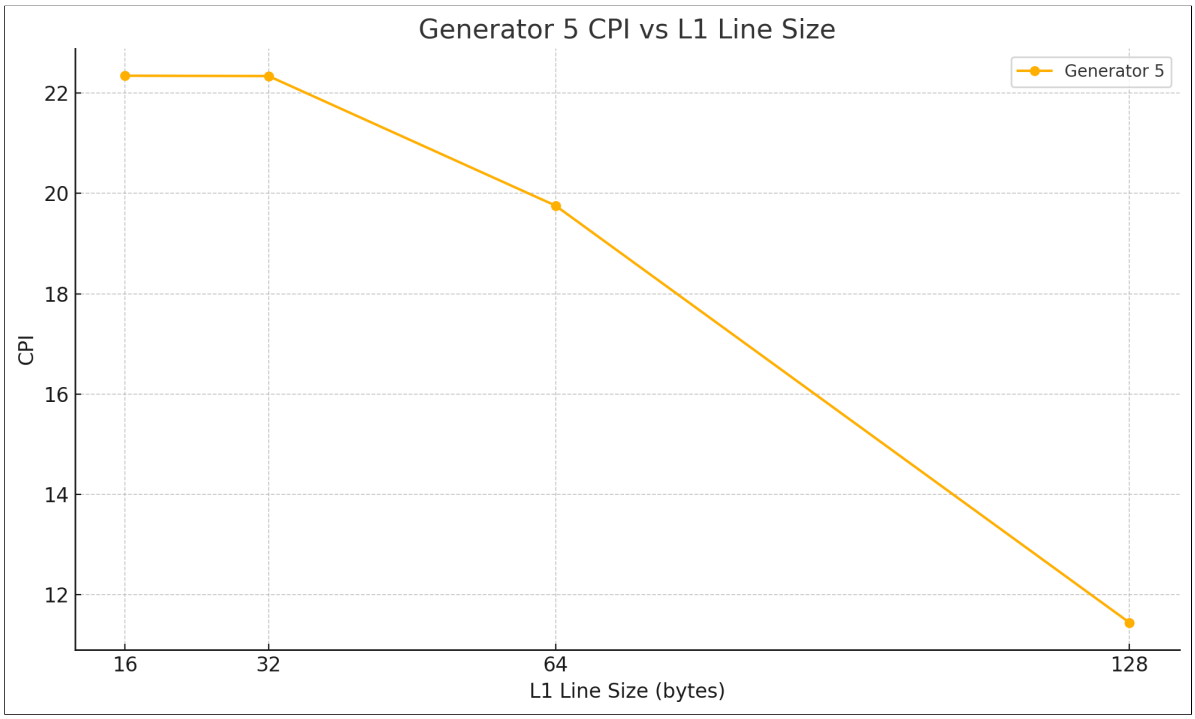


;

Line Size (B)	CPI	L1 Hit Rate	L1 Miss Rate	L2 Hit Rate	L2 Miss Rate
16	1.0058	0.9993	0.0007	0.7500	0.2500
32	1.0045	0.9996	0.0004	0.5000	0.5000
64	1.0038	0.9998	0.0002	0.0000	1.0000
128	1.0019	0.9999	0.0001	0.0000	1.0000

A sequential generation over a 4KB range fits entirely in the 16KB L1, so after the first pass L1 hit rate jumps to nearly 100% and stays there for all line sizes, thus resulting in very low CPI. L2 traffic vanishes after the initial fills (L2 hit rate drops to 0%) making L2 irrelevant.

3.6 Generator 5



Line Size (B)	CPI	L1 Hit Rate	L1 Miss Rate	L2 Hit Rate	L2 Miss Rate
16	22.3478	0.0000	1.0000	0.6419	0.3581
32	22.3420	0.0000	1.0000	0.6420	0.3580
64	19.7580	0.5000	0.5000	0.3944	0.6056
128	11.4423	0.7500	0.2500	0.4442	0.5558

Sequential access with a 32B step means that with 16B or 32B L1 lines will always miss (no two accesses share a line), so L1 miss rate is 100% and CPI stays high. As soon as the line size exceeds the 32B step, consecutive accesses map to the same line in L1, so the hit rate increases, lowering CPI.

4. Conclusion

Through our two-level cache simulator, we have seen how L1 line size interacts with different access patterns to influence overall CPI. For workloads with strong spatial locality (Generator 1 and Generator 4), increasing the L1 line size steadily improved hit rates and significantly decreased the CPI. In contrast, pure randomness over both small (Generator 2) and large address spaces (Generator 3) resulted in near zero L1 hits and rare L2 hits, yielding almost flat CPI curves.

These findings show that bigger cache lines help cut miss costs for sequential and step-based accesses but can hurt performance when they outgrow the next level's block size or when access patterns are random. Keeping the L1 line size at 64B (matching the L2 block size) strikes the best balance, delivering most of the spatial-locality gains without extra write-back overhead.