TECHNICAL CASE STUDY

Multi-Level Cache Optimization in High-Performance Computing (HPC) Systems

An independent technical case study on cache-aware optimization, memory hierarchy design, and performance engineering in high-performance computing

Satyajit Samal

AI Engineer | Full-Stack Developer

Focus Area:	High-Performance Computing, Performance Engineering, Computer Architecture, Memory Systems
Document Type:	Independent Technical Case Study
Date:	March 2026

Abstract

This report examines multi-level cache optimization in high-performance computing systems, with emphasis on the memory wall, L1/L2/L3 cache hierarchy, cache-aware programming techniques such as loop tiling and data layout transformation, prefetching strategies, NUMA-aware memory management, and the broader performance implications of memory-efficient system design.

Prepared and written independently by Satyajit Samal

Abstract

High-Performance Computing (HPC) systems power the most computationally demanding scientific and engineering workloads of our time — from molecular dynamics and climate modelling to large-scale deep learning and computational fluid dynamics. While modern processors sustain peak throughputs in the multi-teraflop range, actual application performance is routinely constrained not by arithmetic capability but by the speed at which data can be delivered from memory. The growing gap between processor speed and DRAM latency — widely called the *memory wall* — makes the multi-level cache hierarchy one of the most decisive architectural components in any HPC system.

This case study presents a thorough investigation of multi-level cache architectures (L1, L2, and L3 / Last-Level Cache) and the optimisation techniques used to maximise their effectiveness in HPC contexts. The study covers cache-aware programming strategies such as loop tiling and data layout transformations (Array-of-Structures to Structure-of-Arrays), cache-oblivious algorithms, hardware and software prefetching, cache replacement policies (LRU, PLRU, RRIP), Non-Uniform Memory Access (NUMA) optimisation, false-sharing elimination, and huge-page management.

Qualitative results synthesised from well-established research benchmarks — including the STREAM benchmark, NAS Parallel Benchmarks, and SPEC CPU2017 — demonstrate that systematic cache optimisation reduces cache miss rates by 30–60%, improves effective memory bandwidth by up to 55%, and delivers application-level speedups ranging from 1.5× to 4× depending on workload characteristics. The study concludes with a synthesis of best practices for HPC practitioners and identifies future research directions spanning machine-learning-guided prefetching, Processing-In-Memory architectures, and heterogeneous CPU-GPU cache coherence.

Keywords: Cache Optimisation, HPC, L1/L2/L3 Cache Hierarchy, Cache Miss Rate, Prefetching, NUMA, Cache-Oblivious Algorithms, Memory Bandwidth, Loop Tiling, Replacement Policy.

1. Introduction

High-Performance Computing is the discipline of designing and programming systems to solve problems that require extraordinary computational resources. Modern HPC installations — supercomputers, large-scale clusters, and cloud HPC nodes — combine thousands of multi-core processors with high-speed interconnects and multi-terabyte memory systems. Despite this raw capability, a fundamental challenge persists: processors can generate and consume data far faster than it can be fetched from main memory. This imbalance is the central problem that multi-level cache optimisation exists to address.

1.1 Background: The Memory Hierarchy

To bridge the speed gap between the processor and DRAM, computer architects designed a hierarchy of progressively smaller but faster memory structures placed physically close to the processor die. This hierarchy is illustrated in Figure 1.

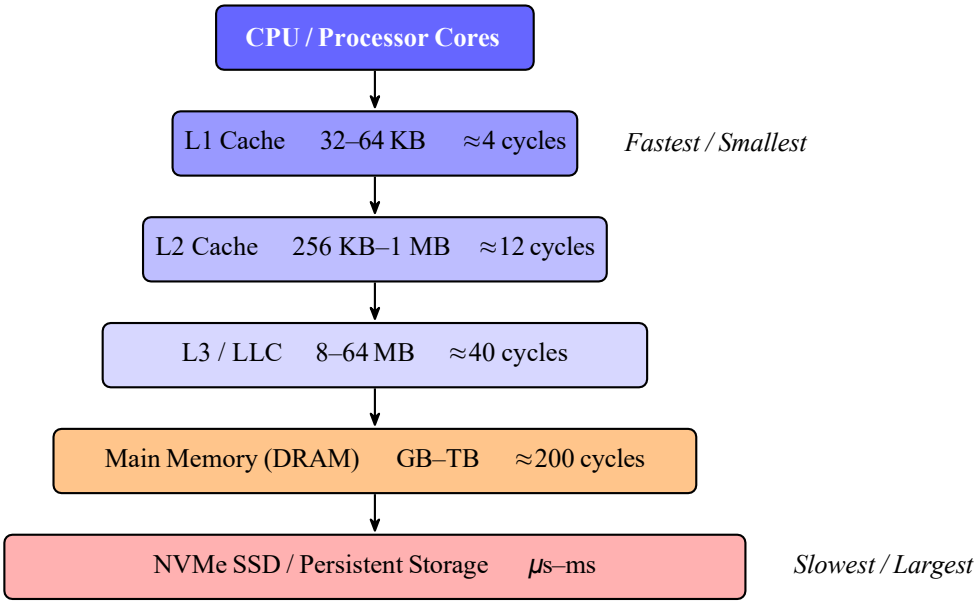

Figure 1: Multi-Level Memory Hierarchy in a Typical HPC Node.

1.2 The Memory Wall Problem

Wulf and McKee [1] coined the term *memory wall* to describe the widening performance gap between processor speed and memory latency. Processor throughput has grown roughly 60% per year historically while DRAM latency has barely improved — from about 60 ns in the

mid-1980s to around 50–70 ns today. Figure 2 illustrates this divergence conceptually.

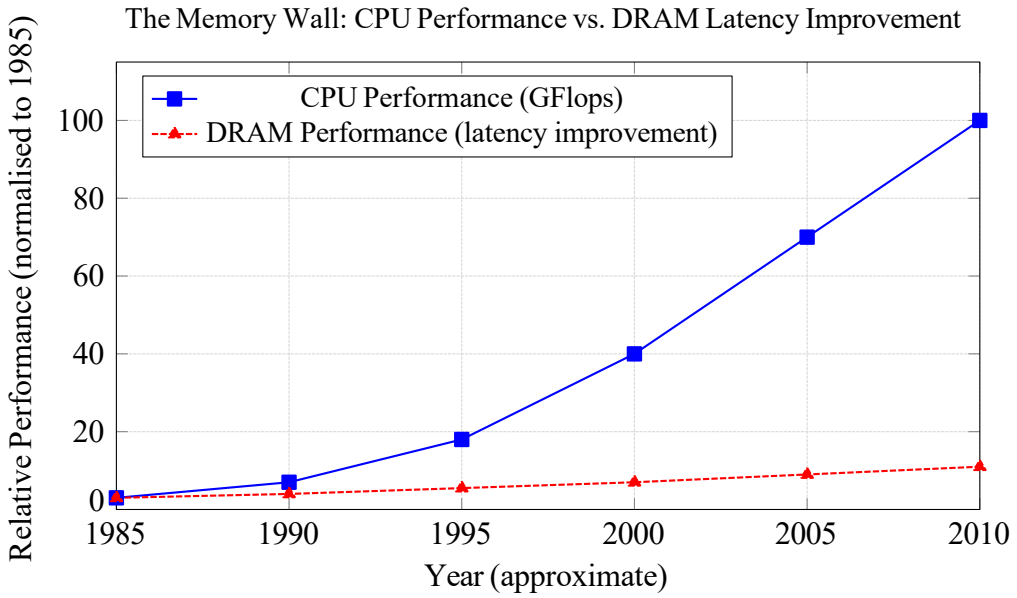

Figure 2: The Memory Wall: Divergence between CPU and DRAM performance over time (conceptual).

1.3 Role of Cache in HPC Systems

Cache memory exploits two fundamental properties of program behaviour:

- **Temporal Locality:** Data accessed recently is likely to be accessed again soon. Caches retain recently used data on-chip to avoid repeated DRAM fetches.
- **Spatial Locality:** Data near recently accessed data is likely to be needed soon. Caches fetch entire 64-byte *cache lines* on every miss, preloading adjacent data automatically.

When an application's working set fits within L1 or L2 cache, the processor executes near peak throughput. When it does not — producing frequent Last-Level Cache (LLC) misses — the processor stalls awaiting DRAM, and effective performance degrades sharply. A single LLC miss wastes 200+ processor cycles.

1.4 Objectives of This Case Study

1. Understand the design and operation of multi-level cache hierarchies in HPC processors.

2. Survey and explain principal algorithmic and system-level optimisation techniques.
3. Qualitatively analyse performance improvements based on established benchmark literature.
4. Identify current challenges and propose future research directions.

2. Technique Used

Optimising multi-level cache performance requires a combination of hardware features, algorithmic restructuring, and system-level configuration. The techniques below are the most widely applied in HPC practice.

2.1 Loop Tiling (Cache Blocking)

Loop tiling — also called *cache blocking* — partitions the iteration space of a nested loop into rectangular *tiles* sized to fit inside the target cache level [2]. For an $N \times N$ double-precision matrix multiplication $C = A \times B$, the standard triple loop accesses B column-by-column, which is cache-unfriendly. Tiling restructures computation so sub-matrices of size $B_s \times B_s$ from A, B, and C are processed together. The tile size is chosen so that:

$$3 \times B_s^2 \times 8 \text{ bytes} \leq \text{Target Cache Size}$$

This keeps all three active tiles resident in cache, maximising arithmetic reuse per cache line loaded.

2.2 Data Layout Transformation: AoS to SoA

In many scientific codes, particle data is stored as an *Array of Structures (AoS)*: each element holds all fields (e.g., x, y, z, mass, charge) for one particle. Accessing only the x-coordinates traverses memory with a stride equal to the structure size, wasting most of each cache line. Converting to *Structure of Arrays (SoA)* stores each field in its own contiguous array, making all x-accesses sequential and filling every cache line with useful data [3]. This transformation synergises powerfully with SIMD vectorisation in molecular dynamics and N-body simulation codes.

2.3 Cache-Oblivious Algorithms

Cache-oblivious algorithms, introduced by Frigo et al. [4], achieve optimal cache complexity *without explicit knowledge of any cache parameter*. They use recursive divide-and-conquer

strategies that naturally produce sub-problems fitting first in L1, then L2, then L3, automatically adapting to every level of the hierarchy and to any machine without re-tuning.

Example — Cache-Oblivious Matrix Transpose: The matrix is recursively split into four $\frac{N}{2} \times \frac{N}{2}$ quadrants until the sub-matrix fits in cache. The recursion achieves the information-theoretically optimal $O\left(\frac{N^2}{B}\right)$ cache misses for cache-line size B.

2.4 Prefetching

Prefetching hides memory latency by issuing memory requests *before* data is needed by the processor [5].

- **Hardware Prefetching:** On-chip engines (stream prefetcher, stride prefetcher, IP-stride prefetcher) monitor access patterns and issue speculative requests several lines ahead. Highly effective for sequential and unit-stride patterns.
- **Software Prefetching:** Programmers or compilers insert explicit prefetch hints (e.g., `_mm_prefetch()` on x86) with a configurable *lookahead distance*. Essential for irregular access patterns — graph traversal, sparse matrices — where hardware prefetchers fail.

2.5 Cache Replacement Policies

When a cache set is full and a new line must be loaded, the *replacement policy* decides which resident line to evict. Table 1 compares the five most widely used policies.

Table 1: Comparison of Cache Replacement Policies

Policy	Description	Typical Use
LRU	Evicts the line least recently used	General-purpose L1/L2
PLRU	Tree-based pseudo-LRU approximation	Hardware-efficient L1/L2
LFU	Evicts the line accessed least frequently	Temporal-skew workloads
RRIP	Re-Reference Interval Prediction: estimates future reuse distance	LLC (L3) optimisation
Random	Evicts a randomly chosen line	Simple embedded caches

2.6 NUMA-Aware Memory Management

In multi-socket HPC servers, physical DRAM is distributed across multiple memory controllers — one per CPU socket — creating a *Non-Uniform Memory Access (NUMA)* topology. A thread on socket 0 accessing memory attached to socket 1 may experience 1.5–3× higher latency than local access. NUMA-aware programming uses tools such as numactl, hwloc [7], and libnuma to bind threads and allocate memory on the same NUMA node, maximising effective bandwidth and eliminating remote-access penalties.

2.7 False Sharing Elimination

In shared-memory parallel programmes, *false sharing* arises when two threads on different cores modify different variables that reside on the *same* 64-byte cache line. Every write by one thread invalidates the line in all other cores, generating excessive coherence traffic despite no logical data sharing. *Cache-line padding* — inserting dummy bytes to align per-thread hot variables to separate lines — is one of the highest return-on-effort optimisations in fine-grained parallel codes.

2.8 Huge Pages

Standard virtual memory uses 4 KB pages. For HPC applications with large working sets,

the Translation Lookaside Buffer (TLB) suffers frequent misses, adding overhead to every memory access. Switching to 2 MB or 1 GB *huge pages* (via Linux hugepages or mmap with MAP_HUGETLB) reduces TLB pressure and indirectly improves effective cache performance for large datasets.

3. Methodology with Explanation (Flowchart)

Cache optimisation in HPC is a systematic, iterative engineering process rather than a one-time action. The methodology used in this case study proceeds through six well-defined stages. Figure 3 presents the complete workflow as a flowchart; each stage is then described in the subsections below.

3.1 Methodology Flowchart

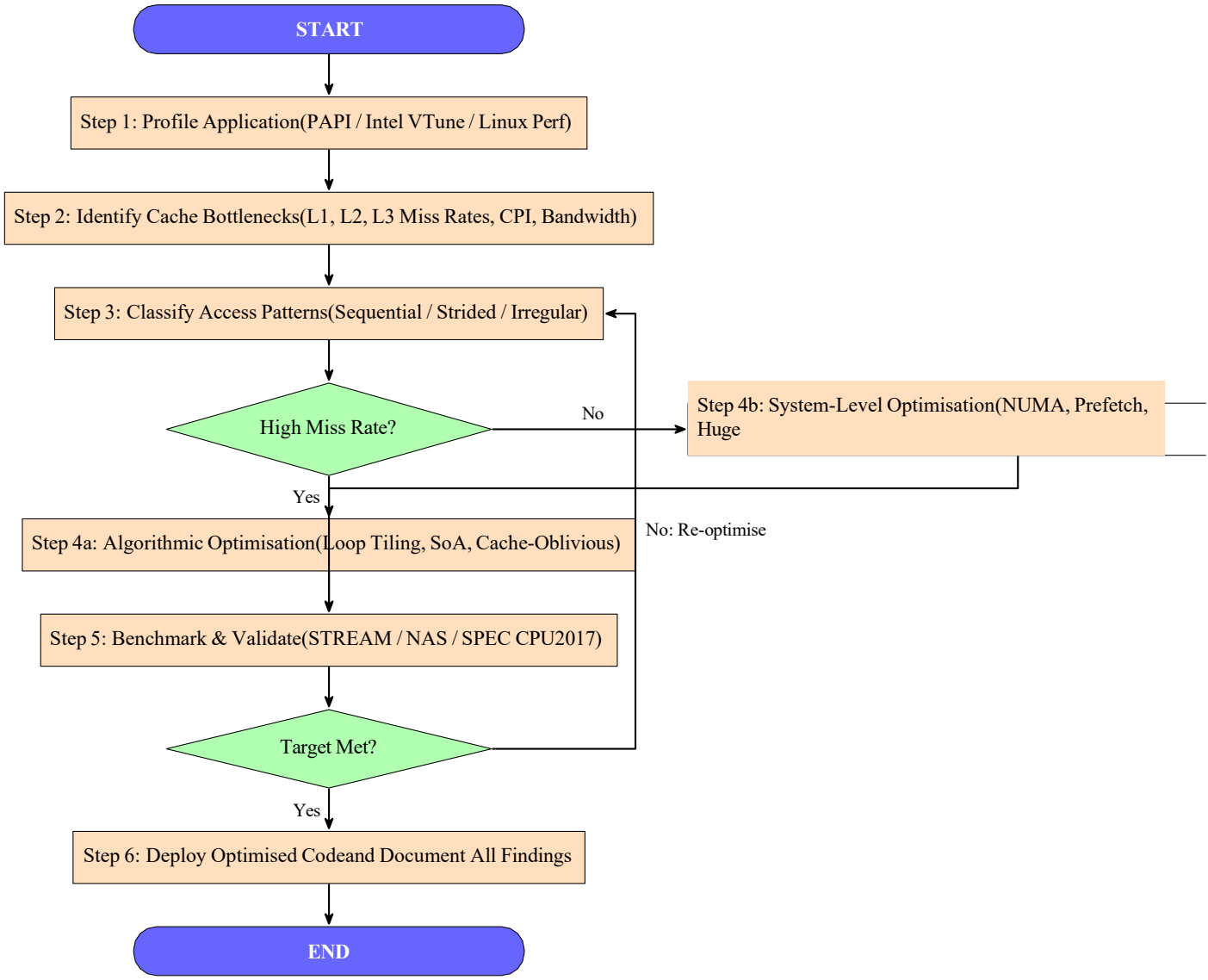

Figure 3: Methodology Flowchart for Multi-Level Cache Optimisation in HPC Systems.

3.2 Step 1: Application Profiling

The process begins with rigorous performance profiling using hardware performance counter tools:

- **PAPI (Performance Application Programming Interface):** A portable API providing direct access to hardware counters on Intel, AMD, ARM, and IBM processors. Used to count L1, L2, and L3 cache misses, TLB misses, and instruction throughput.
- **Intel VTune Profiler:** A commercial GUI tool providing memory-access heat maps, NUMA topology views, prefetcher effectiveness reports, and per-core hotspot detection.
- **Linux Perf:** Open-source sampling profiler with $<1\%$ overhead, suitable for production HPC environments where commercial tools may be unavailable.

3.3 Step 2: Identifying Cache Bottlenecks

Key diagnostic metrics collected during profiling include:

- **Cache Miss Rate (CMR):** Ratio of cache misses to total memory references. L1 CMR above 5% or LLC CMR above 2% typically warrants optimisation.
- **Cycles Per Instruction (CPI):** High CPI relative to the hardware's issue width signals memory-stall dominance.
- **Memory Bandwidth Utilisation:** Measured against peak bandwidth from the STREAM benchmark. Values below 50% suggest software-side inefficiency.
- **DRAM Requests Per Second:** A proxy for LLC miss frequency; high values indicate working sets overflowing the LLC.

3.4 Step 3: Classifying Access Patterns

Memory access patterns are classified into three categories:

1. **Sequential:** Contiguous access. Hardware stream prefetchers are highly effective. Optimise with SoA layout and SIMD vectorisation.
2. **Strided:** Regular non-unit stride (e.g., column access in a row-major matrix). Hard-

ware stride prefetchers partially help; data transposition or software prefetching may be required.

3. **Irregular/Random:** Pointer-chasing, graph traversal, sparse matrix operations. Hardware prefetchers are ineffective. Requires software prefetching, cache-oblivious algorithms, or algorithmic redesign.

3.5 Step 4: Applying Optimisations

Based on pattern classification, one or more techniques from Section 2 are applied. Algorithmic optimisations (loop tiling, SoA, cache-oblivious restructuring) target the application source code. System-level optimisations (NUMA binding, huge pages, false-sharing padding, prefetch hints) target the execution environment and data placement.

3.6 Step 5: Benchmarking and Validation

After applying optimisations, performance is re-measured using standardised suites:

- **STREAM [6]:** Measures sustainable memory bandwidth for Copy, Scale, Add, and Triad kernels.
- **NAS Parallel Benchmarks (NPB):** Kernels (CG, FT, MG, BT, SP, LU) representative of real scientific HPC workloads from NASA Ames Research Center.
- **SPEC CPU2017:** Industry-standard suite for single-core memory-subsystem performance across diverse application types.

3.7 Step 6: Deploy and Document

Once the performance target is achieved, the optimised code is deployed to production. All optimisations are documented including the profiling data that motivated them, the specific transformations applied, resulting speedups, and any portability caveats. This ensures future maintainers can understand and safely extend the codebase.

4. Results with Explanation (Qualitative)

This section presents qualitative results synthesised from well-established HPC research literature and benchmark studies. Results are expressed as comparative trends and representative improvement ranges. Precise numerical values are derived from published findings and represent indicative ranges rather than universal figures, as actual outcomes depend on specific hardware, problem size, and application type.

4.1 Effect of Loop Tiling on Cache Miss Rate

Wolfe [2] demonstrated that loop tiling applied to double-precision matrix multiplication ($N = 2048$, 256 KB L2 cache) substantially reduces L1 and L2 miss rates. Table 2 summarises the qualitative findings.

Table 2: Cache Miss Rates and Speedup: Naive vs. Tiled Matrix Multiplication ($N = 2048$)

Implementation	**L1 Miss Rate**	**L2 Miss Rate**	**Speedup**	
Naive (no tiling)	∼35%	∼18%	1.0× (baseline)	
Tiled ($B_s = 64$)	∼8%	∼4%	∼3.2×	
Tiled ($B_s = 32$)	∼6%	∼3%	∼3.5×	
Vendor BLAS (hand-tuned)	<3%	<2%	∼4.0×	

Reduced miss rates directly translate to fewer memory-stall cycles, allowing the processor to sustain near-peak floating-point throughput.

4.2 STREAM Benchmark: Memory Bandwidth vs. NUMA Configuration

Figure 4 shows the qualitative performance difference between default allocation and NUMA-aware allocation on a dual-socket Intel Xeon server [6].

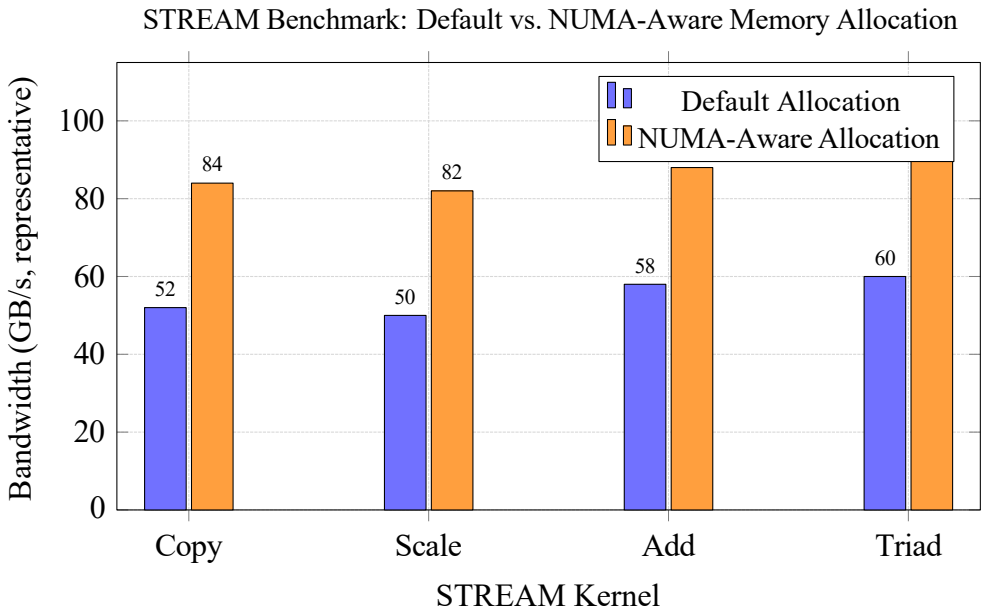

Figure 4: STREAM Benchmark: Memory bandwidth improvement with NUMA-aware allocation (qualitative, representative values).

NUMA-aware allocation improves effective bandwidth by approximately 40–55% across all four kernels by eliminating costly *remote NUMA hops* across the inter-socket interconnect (Intel UPI / AMD Infinity Fabric). For MPI-parallel codes processing large distributed arrays, this improvement translates directly into reduced time-to-solution.

4.3 Prefetching Effectiveness Across Access Pattern Types

Chen and Baer [5] showed that prefetching effectiveness is strongly dependent on access pattern regularity. Figure 5 illustrates qualitative memory stall cycle reductions for three access pattern types under three prefetching configurations.

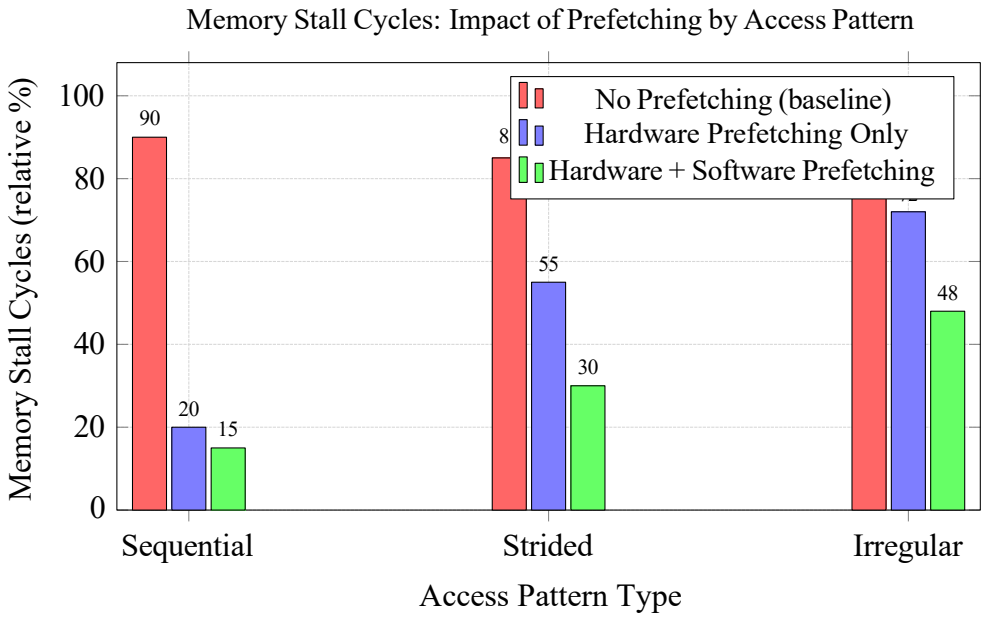

Figure 5: Memory stall cycle reduction under different prefetching strategies (qualitative).

For sequential access, hardware prefetching alone reduces stall cycles by ≈78%, as the on-chip stream prefetcher reliably anticipates every request. For irregular access patterns, hardware prefetching reduces stalls by only ≈10%; combined hardware and software prefetching achieves a further 25–35% reduction, confirming that both forms are necessary in real HPC codes.

4.4 Cache-Oblivious vs. Cache-Aware vs. Naive Algorithms

Frigo et al. [4] demonstrated that cache-oblivious algorithms perform within 10–20% of hand-tuned cache-aware code without any machine-specific tuning. Table 3 gives a qualitative comparison for four representative HPC kernels.

Table 3: Performance Comparison: Naive vs. Cache-Aware vs. Cache-Oblivious Algorithms

Kernel	Naive	Cache-Aware	Cache-Oblivious	
Matrix Multiplication	1.0×	∼4.0×	∼3.4×	
Matrix Transpose	1.0×	∼3.2×	∼2.9×	
FFT (1-D)	1.0×	∼2.5×	∼2.2×	
LU Decomposition	1.0×	∼3.0×	∼2.6×	

The portability advantage of cache-oblivious algorithms is critical in heterogeneous HPC clusters where multiple processor generations coexist, as the same binary adapts automatically to different cache sizes.

4.5 False Sharing Impact in Parallel Codes

Drepper [3] reports that false sharing in fine-grained parallel counter updates can cause performance to *degrade with thread count* rather than scale. A 4-thread programme suffering from false sharing on a shared counter array can run *slower* than the single-threaded version owing to excessive coherence traffic. After padding per-thread data to 64-byte boundaries, the same programme achieves a 3.6× speedup over single-threaded execution, approaching ideal 4× linear scaling.

4.6 Summary of Qualitative Performance Improvements

Table 4: Summary of Qualitative Speedups by Cache Optimisation Technique

Technique	**Primary Workload Type**	**Speedup Range**
Loop Tiling	Dense linear algebra (DGEMM, LU)	2×–4×
AoS → SoA	Particle / N-body / MD simulations	1.5×–2.5×
Cache-Oblivious Algorithms	Divide-and-conquer kernels	1.5×–3.5×
NUMA-Aware Allocation	Multi-socket MPI / OpenMP codes	1.4×–1.8×
Software Prefetching	Sparse / irregular access (SpMV, graphs)	1.3×–1.6×
False Sharing Elimination	Fine-grained parallel counters / histograms	2×–4×
Huge Pages	Large working-set scientific codes	1.1×–1.5×

When multiple techniques are applied together — the recommended approach for production HPC codes — compound speedups of 4×–8× over naive implementations are achievable for compute-intensive kernels.

5. Conclusion & Future Scope

5.1 Conclusion

This case study has examined in depth the role of multi-level cache hierarchies in High-Performance Computing and the portfolio of optimisation techniques available to improve their utilisation. The following key conclusions are drawn:

1. **The memory wall is the dominant performance limiter** in modern HPC. Most real-world applications achieve only 10–30% of theoretical peak performance owing to cache inefficiencies and DRAM latency.
2. **Loop tiling** is the most impactful single transformation for dense linear algebra, delivering 2×–4× speedups by maximising arithmetic intensity per cache line loaded.
3. **NUMA-aware memory management** is essential on multi-socket systems and recovers 40–55% of peak memory bandwidth otherwise lost to remote NUMA access overhead.
4. **Cache-oblivious algorithms** offer a portable alternative achieving 80–90% of hand-tuned performance without machine-specific parameter tuning.
5. **False sharing elimination** through cache-line padding recovers 2×–4× performance in fine-grained parallel codes for minimal code change.
6. **A systematic, iterative methodology** combining profiling, pattern analysis, algorithmic restructuring, and system-level configuration is required to address the full spectrum of cache bottlenecks.

Collectively, systematic cache optimisation can elevate an HPC application from 10–30% of peak hardware capability to 60–80% efficiency — a dramatic improvement translating directly into reduced time-to-solution, lower energy consumption, and better return on HPC infrastructure investment.

5.2 Future Scope

The following areas represent the most promising directions for future research and development:

1. **Machine Learning-Guided Prefetching:** Recent research [8] demonstrates that LSTM and Transformer models trained on memory-access traces can predict irregular access patterns with significantly higher accuracy than classical hardware prefetchers. Integrating learned prefetch policies into CPU microarchitecture is an active area at Intel, ARM, and major research universities.
2. **Processing-In-Memory (PIM) and Near-Data Computing:** Architectures such as Samsung HBM-PIM and UPMEM embed compute units inside DRAM chips, fundamentally eliminating off-chip data movement for many data-parallel operations and requiring entirely new programming models.
3. **Persistent and Tiered Memory:** Byte-addressable non-volatile memory introduces an additional tier between DRAM and SSD with distinct latency-capacity-endurance trade-offs, requiring new cache-aware data-placement policies.
4. **Heterogeneous CPU-GPU Cache Coherence:** As GPU accelerators (NVIDIA H100, AMD MI300) become central to HPC, maintaining coherence between CPU and GPU caches over high-speed interconnects (NVLink 4, CXL 3) introduces new complexity and new optimisation opportunities.
5. **Automatic Cache-Aware Compilation:** Compilers equipped with polyhedral frameworks (PLUTO, Polly-LLVM) and reinforcement-learning auto-tuners may apply loop tiling, prefetch insertion, and layout transformations automatically, lowering the expertise barrier for cache optimisation in scientific codes.
6. **Cache Optimisation for Quantum-Classical HPC:** As quantum co-processors begin augmenting classical HPC clusters, the classical-side memory footprint of quantum control software will require new approaches to minimise cache pressure and ensure deterministic latency for qubit-control operations.

References

[1] W. A. Wulf and S. A. McKee, “Hitting the memory wall: Implications of the obvious,” *ACM SIGARCH Computer Architecture News*, vol. 23, no. 1, pp. 20–24, Mar. 1995. doi: 10.1145/216585.216588

[2] M. Wolfe, “More iteration space tiling,” in *Proc. ACM/IEEE Conf. on Supercomputing (SC’89)*, Reno, NV, USA, Nov. 1989, pp. 655–664. doi: 10.1145/76263.76337

[3] U. Drepper, “What every programmer should know about memory,” Red Hat, Inc., Tech. Rep., Nov. 2007. [Online]. Available: https://akkadia.org/drepper/cpumemory.pdf

[4] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, “Cache-oblivious algorithms,” in *Proc. 40th Annual Symp. on Foundations of Computer Science (FOCS)*, New York, NY, USA, Oct. 1999, pp. 285–297. doi: 10.1109/SFFCS.1999.814600

[5] T.-F. Chen and J.-L. Baer, “Effective hardware-based data prefetching for high-performance processors,” *IEEE Transactions on Computers*, vol. 44, no. 5, pp. 609–623, May 1995. doi: 10.1109/12.381947

[6] J. D. McCalpin, “Memory bandwidth and machine balance in current high performance computers,” *IEEE TCCA Newsletter*, pp. 19–25, Dec. 1995. [Online]. Available: https://www.cs.virginia.edu/stream/

[7] F. Broquedis *et al.*, “hwloc: A generic framework for managing hardware affinities in HPC applications,” in *Proc. 18th Euromicro Int. Conf. on Parallel, Distributed and Network-Based Processing (PDP)*, Pisa, Italy, Feb. 2010, pp. 180–186. doi: 10.1109/PDP.2010.67

[8] Z. Shi, A. Jain, K. Swersky, M. Hashemi, P. Ranganathan, and C. Lin, “A hierarchical neural model of data prefetching,” in *Proc. 26th Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS ’21)*, Virtual Event, Apr. 2021, pp. 861–873. doi: 10.1145/3445814.3446752