Data-Oriented vs. Object-Oriented Design: A Performance Comparison

Pratik Pujari, Pimpisut Puttipongkawin
University of Colorado Boulder
{pratik.pujari, pimpisut.puttipongkawin}@colorado.edu

Abstract

This paper presents a comparative analysis of Object-Oriented Design (OOD) and Data-Oriented Design (DOD) paradigms, focusing on their performance implications in compute-intensive applications. While OOD emphasizes encapsulation, inheritance, and abstraction to achieve modularity and maintainability, DOD prioritizes memory layout optimization and data locality to leverage modern CPU cache hierarchies. We evaluate both paradigms through two experimental implementations—a particle simulation system and an array processing task—measuring execution time, cache efficiency, and memory access patterns. The study offers empirical insights into when performance-critical applications may benefit from data-oriented approaches over traditional object-oriented design.

Introduction

Modern software development must balance readability, maintainability, and performance. *Object-Oriented Design (OOD)* has long served as a foundation for building modular systems through encapsulation, inheritance, and polymorphism [9]. However, OOD's emphasis on abstraction and object encapsulation can result in fragmented memory access and inefficient use of modern CPU caches [9].

To address these limitations, *Data-Oriented Design (DOD)* focuses on how data is structured and accessed in memory rather than how objects interact. By prioritizing data locality, DOD enables more efficient cache utilization and better alignment with processor behavior. A central DOD concept is the shift from an *Array of Structures (AoS)* to a *Structure of Arrays (SoA)* layout, ensuring contiguous memory access and improved spatial locality [6]. This approach has been widely adopted in performance-critical domains such as game development [8, 1] and high-performance computing.

This paper compares OOD and DOD through two experimental implementations—a particle simulation and an array processing system—to analyze their effects on execution time, cache efficiency, and memory access behavior.

Background

Object-Oriented Design Paradigm

Object-Oriented Design (OOD) has become the dominant paradigm in enterprise software through its core principles of encapsulation, inheritance, and polymorphism [9]. These features enable developers to manage complexity through abstraction and hierarchical organization, making OOD highly effective for building maintainable, modular systems.

However, this abstraction comes at a cost. When numerous small objects are distributed across non-contiguous memory regions, the CPU must perform frequent memory fetches, each potentially causing cache misses. Fedoseev et al. [5] observe that while OOD excels in maintainability, its object-per-entity model creates performance bottlenecks in data-intensive scenarios such as game engines and scientific simulations. The indirection inherent in polymorphic designs further compounds this issue, as virtual function calls introduce additional memory lookups and prevent compiler optimizations [7].

Memory Hierarchy and Cache Behavior

Modern CPU performance increasingly depends on effective cache utilization. With L1 cache operating at nanosecond speeds while main memory requires hundreds of nanoseconds, the performance gap between processors and memory continues to widen [4]. This growing disparity—often called the "memory wall"—means that cache-oblivious algorithms can suffer severe performance penalties regardless of their computational complexity.

Key factors affecting cache performance:

- **Spatial locality:** Sequential memory access enables hardware prefetching mechanisms to predict and preload data
- **Temporal locality:** Recently accessed data stays cache-resident, reducing redundant memory fetches
- **Cache line utilization:** Modern processors fetch 64-byte blocks; underutilized lines waste bandwidth

Drepper [4] demonstrates that cache-aware programming can produce 2-10× speedups in memory-bound applications. Chilimbi et al. [3] showed that reorganizing data structures to group frequently accessed fields within cache blocks reduces miss rates by 20-40% without algorithmic changes.

Their work on cache-conscious structure layout pioneered automated profiling tools that identify optimization opportunities through runtime analysis.

In heterogeneous computing environments, memory scheduling strategies become even more critical. Ausavarungnirun et al. [2] demonstrated that intelligent memory scheduling can improve system throughput by 19% and fairness by 31% in multi-core systems, showing that data access patterns matter not just within individual applications but across entire computing systems.

Data-Oriented Design Principles

Data-Oriented Design (DOD) rethinks software organization by aligning data layout with hardware characteristics rather than object hierarchies. The paradigm focuses on ”how data flows through computations” instead of ”what objects do,” explicitly optimizing for cache behavior and memory bandwidth [1].

Core DOD techniques:

- **Structure of Arrays (SoA):** Replace object arrays with attribute arrays to maximize sequential access
- **Sequential processing:** Operate on complete arrays before moving to next attribute, leveraging prefetching
- **Minimal indirection:** Reduce pointer chasing and virtual function calls that disrupt cache prediction
- **Hot/cold data splitting:** Separate frequently accessed fields from rarely used ones [8]

Mironov et al. [6] demonstrated a 28% execution time improvement using SoA layouts, reducing cache misses by 35% in a trading backtester. The study emphasized that DOD's benefits emerge not just from better cache hit rates but from reduced memory traffic overall—fewer bytes transferred means less time spent waiting on memory subsystems.

Nyberg [7] validated these principles on modern Intel architectures, finding that contiguous layouts and reduced polymorphism measurably improve both execution time and cache efficiency. His work showed that replacing virtual function calls with direct data access reduced instruction cache misses and eliminated branch prediction penalties—hidden costs that OOD designs often overlook.

Wingqvist et al. [10] extended this to multi-threaded environments, showing DOD's benefits compound with parallelism—achieving 40% single-threaded improvements and 60-80% gains across multiple cores. The structure-of-arrays layout facilitates better thread-level parallelism by reducing false sharing, where multiple threads inadvertently compete for the same cache lines. This scalability advantage makes DOD particularly attractive for modern multi-core processors.

Trade-offs and Applicability

Despite performance advantages, DOD involves trade-offs. The low-level memory focus can reduce code readability and complicate maintenance—adding attributes requires modifying multiple arrays rather than single class definitions. Nystrom [8] notes that DOD can make code harder to reason about for developers accustomed to object-oriented thinking, as the logical grouping of related data becomes less explicit.

The choice between paradigms depends on application priorities:

- **Favor OOD:** Complex object interactions, runtime polymorphism, collaborative development, evolving requirements
- **Favor DOD:** Predictable data access, performance-critical loops, large homogeneous datasets, real-time constraints

This study implements both paradigms in comparable workloads to empirically evaluate their effects on cache efficiency, execution time, and memory access patterns under controlled conditions.

Related Work

Research comparing Object-Oriented Design (OOD) and Data-Oriented Design (DOD) has expanded in recent years as developers seek to align software structure with modern hardware characteristics. Early empirical studies in game development demonstrated that DOD implementations can achieve substantial speedups over OOD by improving cache coherence and memory access patterns [10, 7]. These findings were largely attributed to the use of contiguous data layouts and reduced object indirection, allowing more predictable memory access and better utilization of CPU pipelines.

Beyond domain-specific applications, studies in high-performance computing and systems design have examined cache-aware and locality-optimized programming techniques. Chilimbi et al. [3] pioneered the concept of cache-conscious data layouts, showing that reorganizing structures to group frequently accessed elements within the same cache block could dramatically reduce cache misses. More recent work extends this idea to modern architectures, emphasizing data-oriented principles such as the *Structure of Arrays* (SoA) pattern and SIMD-friendly processing pipelines. These optimizations are increasingly integrated into performance-critical libraries, including physics engines and numerical solvers.

Acton's influential presentation on data-oriented design [1] highlighted practical considerations for game developers, emphasizing that ”data is all there is” and encouraging programmers to think about transformations on data streams rather than object behaviors. This philosophy has influenced modern game engine architectures, where Entity-Component-System (ECS) patterns naturally align with DOD principles by separating data from behavior.

Nystrom [8] provides practical guidance on data locality patterns in game programming, demonstrating how grouping hot data can dramatically improve cache performance. His work bridges the gap between theoretical computer architecture and practical software engineering, showing concrete examples of how memory layout affects real-world performance.

At the same time, researchers such as Fedoseev et al. [5] have highlighted the trade-offs between performance and maintainability. While DOD provides clear computational benefits, its low-level focus can make systems harder to extend and reason about, particularly in large-scale or collaborative software projects. This tension between abstraction and efficiency motivates further exploration of when DOD provides sufficient performance advantages to justify its complexity.

Despite these insights, existing studies tend to focus on specific application domains, such as gaming or simulation. Few works systematically evaluate DOD and OOD across controlled, hardware-aware benchmarks using modern profiling tools. This paper addresses that gap by quantitatively comparing both paradigms through two compute-intensive experiments, analyzing their performance in terms of execution time, cache efficiency, and memory access behavior.

Methodology

This study consists of two experimental programs, each implemented in both OOD and DOD paradigms.

Particle Physics Simulation

We implemented a 3D particle physics engine in two paradigms to evaluate memory access patterns under compute-intensive workloads. The simulation models 100,000 particles over 1,000 iterations with gravitational forces and boundary collisions.

OOD Implementation: Each particle is represented as an object containing nine floating-point attributes (position (x, y, z), velocity (v_x, v_y, v_z), and acceleration (a_x, a_y, a_z)). Updates iterate through the particle array, accessing all attributes sequentially per particle. This follows the traditional Array-of-Structures (AoS) pattern common in object-oriented systems.

DOD Implementation: Particle data is decomposed into nine separate arrays (Structure of Arrays). Each update phase processes entire arrays sequentially—first all v_x values, then all v_y values, and so forth. This maximizes spatial locality for each operation and enables more effective hardware prefetching.

Both implementations use identical algorithms to ensure numerical equivalence (profiling methodology in Metrics and Tools). The complete source code, build environment, and profiling scripts are available at https://github.com/bugswap/OOD-DOD.

Array Processing Task

[Work in progress] This experiment will process large numerical arrays with arithmetic operations to further validate the performance characteristics observed in the particle simulation. The OOD version will encapsulate each element in an object with associated methods, while the DOD version will process contiguous data arrays directly. We plan to measure cache behavior and execution time using the same profiling methodology described in Metrics and Tools.

Metrics and Tools

Performance was measured using GNU `time` for execution timing (wall-clock, CPU utilization, memory usage) and Valgrind `cachegrind` 3.18.1 for cache profiling. Cachegrind simulated a 16KB L1 data cache (64-byte lines, 4-way associative) and 256KB last-level cache (8-way associative), recording instruction counts, data reads/writes, and cache miss rates by operation type.

Programs compiled with `g++ -O3 -std=c++17` executed in a Docker container (Ubuntu 22.04) for reproducibility. Each test ran 1,000 iterations over 100,000 particles with identical initialization (seed=42), producing numerically equivalent outputs (error $< 10^{-5}$). All measurements represent averages of three runs with standard deviation <2%.

Table 1: Performance and Cache Metrics Comparison

Metric	**OOD**	**DOD**
Execution		
Time (ms)	250	202
Speedup	1.0×	1.24×
Memory (KB)	6,324	6,460
Memory Access (Millions)		
Total References	1,122	816
Data Reads	701	620
Data Writes	421	196
Instructions	3,669	3,613
L1 Data Cache (16KB)		
Overall Miss Rate	10.0%	13.0%
Read Miss Rate	8.0%	16.0%
Write Miss Rate	13.4%	3.7%
Last-Level Cache (256KB)		
Overall Miss Rate	10.0%	12.9%
Write-backs (M)	56.3	6.4

Results and Discussion

Performance Comparison

Table 1 shows the execution time and cache behavior for both implementations. DOD ran 19.2% faster (202ms vs 250ms) and reduced total memory operations by 37.5% (816M vs 1,122M references). Both implementations produced identical numerical results, confirming they use the same algorithm.

Cache Behavior Analysis

Surprisingly, DOD had higher L1 miss rates (13.0% vs 10.0%) but still ran faster. This shows that DOD's performance advantage comes from three mechanisms, not just better cache hits:

Memory bandwidth efficiency: OOD's layout loads entire 36-byte particle structures even when only 12 bytes are needed for specific operations. DOD's structure-of-arrays avoids this waste, reducing total memory operations by 37.5%. This follows cache-conscious design principles that improve spatial locality [3]. The reduced bandwidth consumption aligns with observations by Ausavarungnirun et al. [2] that efficient memory scheduling and reduced traffic can significantly improve system-wide performance.

Write optimization: OOD's write miss rate (13.4%) is much higher than DOD's (3.7%), showing frequent cache evictions. DOD reduced write-backs by 88.6% (56.3M vs 6.4M) because its sequential array writes use hardware write-combining buffers that group stores together and reduce cache pollution [4]. This optimization is particularly important in write-heavy workloads, as Drepper notes that write-through caches can become bottlenecks when eviction rates are high.

Hardware prefetching: DOD's sequential array access lets the CPU's prefetcher predict and load data before it is needed, hiding memory delays through parallel operations. OOD's scattered access pattern prevents effective prefetching, causing operations to wait for full memory latency. Modern CPUs are designed to exploit these regular patterns [4], and Nystrom [8] emphasizes that predictable access patterns are essential for maximizing hardware performance in data-intensive applications.

Implications

These results show that DOD works by reducing memory traffic (37.5%) and enabling hardware optimizations, not by achieving better cache hit rates. Each attribute array uses 400KB of memory, which is much larger than the 16KB L1 cache. This causes DOD's higher miss rates, but these misses happen in patterns that hardware can optimize.

DOD works best for: (1) large datasets that exceed cache size, (2) operations that only need some object attributes, and (3) systems with strong prefetching. The 19.2% speedup from 37.5% less memory traffic shows that memory bandwidth, not just cache hits, is the main bottleneck in data-heavy applications. This matches earlier studies showing DOD's performance benefits in compute-intensive tasks [7, 10, 6].

The findings also suggest that DOD's advantages would compound in multi-threaded scenarios, as reduced memory traffic would decrease bus contention between cores [10]. Future work should explore these scaling characteristics on modern multi-core architectures.

Conclusion

This research demonstrates that DOD's performance advantages stem from reduced memory bandwidth consumption and hardware optimization enablement rather than superior cache hit rates alone. The particle simulation revealed a 19.2% speedup despite higher L1 miss rates, achieved through 37.5% fewer memory operations and 88.6% reduction in write-backs.

DOD proves most effective for large datasets exceeding cache capacity where operations access attribute subsets rather than complete objects. The structure-of-arrays pattern enables hardware prefetchers to hide memory latency and reduces unnecessary data transfer—benefits that become increasingly important as the gap between CPU and memory speed widens [4].

While OOD remains preferable for maintainable, modular systems requiring complex object interactions and runtime polymorphism [9, 5], DOD provides measurable benefits in data-intensive applications with predictable access patterns. The choice between paradigms should be guided by application requirements: systems prioritizing flexibility favor OOD, while performance-critical workloads with homogeneous data benefit from DOD.

Future work should explore DOD's scalability in multi-threaded environments [10], its interaction with emerging memory architectures, and hybrid approaches that balance the maintainability of OOD with the performance of DOD for different subsystems within a single application.

References

[1] Mike Acton. Data-oriented design and c++. *CppCon*, 2014. Keynote presentation.

[2] Rachata Ausavarungnirun, Kevin Kai-Wei Chang, Lavanya Subramanian, Gabriel H. Loh, and Onur Mutlu. Staged memory scheduling: Achieving high performance and scalability in heterogeneous systems. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA)*, pages 416–427, Portland, OR, USA, 2012. IEEE.

[3] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Cache-conscious structure layout. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 1–12, Atlanta, GA, USA, 1999. ACM.

[4] Ulrich Drepper. *What Every Programmer Should Know About Memory*. Red Hat, Inc., 2007.

[5] Konstantin Fedoseev, Nursultan Askarbekuly, Elina Uzbekova, and Manuel Mazzara. A case study on object-oriented and data-oriented design paradigms in game development. Technical report, Blekinge Institute of Technology, 2020.

[6] Timur Mironov, Leonid Motaylenko, and Dmitry Andreev. Comparison of object-oriented programming and data-oriented design for implementing trading strategies backtester. In *Proceedings of the Information Technology and Management Science Conference*, pages 1–6, Riga, Latvia, 2021. University of Latvia.

[7] Filip Nyberg. Investigating the effect of implementing data-oriented design principles on performance and cache utilization. Bachelor's thesis, Umeå University, Department of Computing Science, 2021.

[8] Robert Nystrom. *Game Programming Patterns*. Genever Benning, 2014. Chapter on Data Locality.

[9] Nehul Singh, Satyendra Singh Chouhan, and Karan Verma. Object oriented programming: Concepts, limitations and application trends. In *Proceedings of the 2021 5th International Conference on Information Systems and Computer Networks (ISCON)*, pages 1–6, GLA University, Mathura, India, 2021. IEEE.

[10] Daniel Wingqvist, Filip Wickström, and Sead Memeti. Evaluating the performance of object-oriented and data-oriented design with multi-threading in game development. In *Proceedings of the 2022 IEEE Games, Entertainment, Media Conference (GEM)*, pages 1–8, St. Michael, Barbados, 2022.