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

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

GitHub Repository: https://github.com/bug-swap/OOD-DOD

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

We implemented a numerical array processing system to evaluate memory access patterns under mixed computational workloads. The system processes 1,000,000 data elements over 500 iterations, performing arithmetic operations (multiplication, square root, sine), scaling transformations, maximum value detection, and normalization.

OOD Implementation: Each data element is encapsulated in a `DataElement` class containing three floating-point attributes (`value`, `coefficient`, `result`) and methods for computation, scaling, normalization, and squared value calculation. Operations iterate through the object array multiple times—once for each computational phase—following the Array-of-Structures pattern.

DOD Implementation: Data is decomposed into three separate arrays using the Structure of Arrays pattern. Each processing phase operates on complete arrays sequentially, maximizing spatial locality. The `compute()` method processes all values before moving to coefficients, enabling hardware prefetchers to load data predictively.

Both implementations use identical algorithms with the same random seed (42) to ensure numerical equivalence (error $< 10^{-5}$). The workload combines math-intensive operations with memory-intensive transformations to evaluate performance under diverse computational patterns.

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 with identical initialization (seed=42), producing numerically equivalent outputs (error $< 10^{-5}$). All measurements represent averages of three runs with standard deviation $<2\%$.

Results and Discussion

Particle Simulation Performance

Table 1 shows the execution time and cache behavior for both implementations. DOD ran 18.5% faster (205ms vs 243ms) and reduced total memory operations by 27.3% (816M vs 1,122M references). Both implementations produced identical numerical results, confirming they use the same algorithm.

Table 1: Particle Simulation Performance and Cache Metrics

Metric	OOD	DOD
Execution		
Time (ms)	243	205
Speedup	1.0×	1.19×
Memory (KB)	6,328	6,396
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

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 27.3%. The reduced bandwidth consumption demonstrates that memory traffic volume matters as much as cache hit rates for overall 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. This optimization is particularly important in write-heavy workloads where cache eviction rates become bottlenecks.

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 regular access patterns for maximum performance.

Array Processing Performance

Table 2 presents performance metrics for the array processing experiment. Unlike the particle simulation, this workload showed different cache characteristics while maintaining DOD's performance advantages.

Table 2: Array Processing Performance and Cache Metrics

Metric	OOD	DOD
Execution		
Time (ms)	1,133	1,098
Speedup	1.0×	1.03×
Memory (KB)	14,652	14,724
Memory Access (Millions)		
Total References	5,992	5,992
Data Reads	4,486	4,735
Data Writes	1,506	1,257
Instructions	28,515	26,260
L1 Data Cache (16KB)		
Overall Miss Rate	6.3%	4.2%
Read Miss Rate	7.7%	4.6%
Write Miss Rate	2.1%	2.5%
Last-Level Cache (256KB)		
Overall Miss Rate	6.3%	4.2%
Data Misses (M)	375.2	250.4

Modest Performance Gains. DOD achieved only a 3.1% speedup (1,098ms vs 1,133ms), significantly smaller than the particle simulation's 18.5% improvement. This reduced advantage stems from the workload's arithmetic intensity—both implementations spend substantial time in transcendental function calls (`sin`, `sqrt`) that dominate execution time regardless of memory layout. Cachegrind profiles reveal that `sinf` operations alone consumed 33% of total instructions, creating a compute-bound bottleneck where memory optimizations have limited impact.

Improved Cache Behavior. Contrary to the particle simulation, DOD demonstrated *better* cache hit rates than OOD. The L1 miss rate decreased from 6.3% to 4.2%, with read miss rates improving from 7.7% to 4.6%. This 33% reduction in cache misses shows that for compute-intensive workloads with smaller element sizes (12 bytes vs 36 bytes in particles), DOD's sequential access achieves both better locality *and* reduced miss rates. The improved cache behavior reduced last-level cache misses by 33.3% (250.4M vs 375.2M).

Instruction Efficiency. DOD reduced instruction count by 7.9% (26.3B vs 28.5B instructions), demonstrating that structure-of-arrays layouts enable more efficient code generation. The compiler can better optimize sequential array operations compared to object method calls, reducing loop overhead and improving instruction-level parallelism.

Memory Traffic Reduction. DOD reduced data writes by 16.5% (1,257M vs 1,506M) while slightly increasing reads (4,735M vs 4,486M). The write reduction indicates more efficient use of write-combining buffers and reduced cache pollution from store operations.

Cross-Experiment Analysis

The contrasting results between particle simulation and array processing reveal that DOD's benefits depend critically on workload characteristics:

Memory-Bound vs. Compute-Bound Workloads. The particle simulation (18.5% speedup) is memory-bound with simple arithmetic operations, where DOD's reduced memory traffic (27.3%) directly translates to performance gains. The array processing task (3.1% speedup) is compute-bound, dominated by transcendental functions consuming 33% of instructions that mask memory access improvements. This demonstrates that DOD provides maximum benefit when memory bandwidth—not computation—is the bottleneck.

Cache Behavior Patterns. The particle simulation showed DOD achieving higher miss rates (13.0% vs 10.0%) but better performance through reduced memory traffic and effective prefetching. The array processing task showed DOD achieving *better* cache hit rates (4.2% vs 6.3%), suggesting that workloads with moderate data sizes relative to cache capacity benefit from both reduced traffic and improved locality.

Data Structure Size Effects. Particle objects (36 bytes) exceed half a cache line (64 bytes), causing partial cache line utilization in OOD. Array elements (12 bytes) are smaller, allowing multiple elements per cache line and reducing OOD's memory waste. The 3× size difference between structures correlates with a 6× difference in speedup (18.5% vs 3.1%), suggesting DOD's advantages scale with object size.

Instruction-Level Differences. DOD reduced instructions by 1.5% in particle simulation but 7.9% in array pro-

cessing, indicating that compiler optimizations favor DOD more in compute-intensive loops. The sequential array access pattern enabled better loop unrolling and vectorization opportunities.

Write Operation Advantages. Both experiments demonstrated dramatic write performance improvements with DOD. The particle simulation reduced write operations by 53.5% and write-backs by 88.6%. The array processing task reduced writes by 16.5%. This consistency suggests DOD's sequential write patterns consistently leverage hardware write-combining buffers more effectively than OOD's scattered writes.

Discussion

Our results align with prior research while revealing nuanced insights into DOD's performance mechanisms. Wingqvist et al. [10] reported similar performance improvements in game development contexts, though their multi-threaded experiments showed larger gains (40-80%) than our single-threaded implementations. This suggests DOD's advantages compound with parallelism.

The particle simulation's counterintuitive result—higher miss rates with better performance—challenges common assumptions about cache optimization. While Drepper [4] emphasizes cache-aware programming, our findings demonstrate that memory bandwidth efficiency and prefetching effectiveness can outweigh raw cache hit rates. This aligns with Ausavarungnirun et al. [2]'s observation that memory scheduling and traffic reduction significantly impact system-wide performance.

The array processing task's modest gains echo Fedoseev et al. [5]'s findings that DOD's benefits diminish in compute-intensive scenarios. However, our 7.9% instruction reduction reveals an underexplored DOD advantage: improved compiler optimization opportunities.

The consistency of write optimization across both experiments validates Chilimbi et al. [3]'s cache-conscious structure layout principles. Nystrom [8]'s practical guidance on data locality patterns finds empirical support in our write performance measurements.

Implications

Our experimental results reveal that DOD's effectiveness depends on the interaction between workload characteristics, data structure design, and hardware behavior.

When to Choose Data-Oriented Design

DOD provides measurable performance benefits under specific conditions:

Memory-Bound Applications. DOD excels when memory bandwidth limits performance. The particle simulation's 18.5% speedup demonstrates DOD's strength where simple operations dominate and memory access patterns determine throughput. Applications spending $>$50% of execution time waiting on memory represent ideal DOD candidates.

Large Object Structures. Performance gains scale with object size. The particle simulation's 36-byte structures showed 6$\times$ greater speedup than the array task's 12-byte elements. When objects exceed cache line size and workflows access individual fields independently, DOD's attribute separation prevents loading unnecessary data.

Predictable Access Patterns. DOD leverages hardware prefetchers through sequential array traversal. Applications with regular, forward-marching access patterns—simulation updates, batch processing, stream transformations—maximize prefetching benefits.

When to Choose Object-Oriented Design

OOD remains preferable when abstraction, maintainability, and flexibility outweigh performance:

Complex Object Interactions. Applications requiring rich object relationships, polymorphic behavior, and runtime type dispatch benefit from OOD's encapsulation.

Evolving Requirements. DOD's low-level memory focus makes structural changes costly. OOD localizes changes within class definitions, enabling incremental evolution without system-wide refactoring.

Compute-Bound Applications. The array processing task's modest 3.1% speedup demonstrates that DOD provides limited benefit when computation dominates execution time.

Conclusion

This research demonstrates through two contrasting experiments that Data-Oriented Design's performance benefits depend critically on workload type. The memory-bound particle simulation achieved 18.5% speedup through 27.3% memory traffic reduction, while the compute-bound array processing task showed only 3.1% improvement despite better cache hit rates. DOD provides maximum benefit when memory bandwidth, rather than computation, limits performance.

Our findings challenge the assumption that cache hit rates directly predict performance. The particle simulation ran faster with *higher* miss rates (13.0% vs 10.0%) through reduced memory traffic, optimized write operations (88.6% fewer write-backs), and effective hardware prefetching. This demonstrates that memory bandwidth efficiency matters more than raw cache hit rates.

DOD proves most effective for memory-bound workloads with large data structures, predictable access patterns, and write-heavy operations. OOD remains preferable for compute-bound applications, complex object interactions, and evolving requirements. Future work should explore DOD's scalability in multi-threaded environments [10] and hybrid approaches that apply DOD selectively to performance-critical subsystems. As the gap between processor and memory speeds continues to widen [4], understanding when to apply DOD principles allows developers to make informed trade-offs between maintainability and performance.

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.