IJCRT.ORG ISSN : 2320-2882

INTERNATIONAL JOURNAL OF CREATIVE RESEARCH THOUGHTS (IJCRT)

An International Open Access, Peer-reviewed, Refereed Journal

A Cache-Centric Performance Analysis Of Pointer-Based Data Structures: Skiplists Versus Linked Lists

[1]Dr. Vijay Kumar Samyal, [2]Raghav Baghla, [2]Aaksh, [2]Chanderkant

[1] Associate Professor, Dept. of CSE MIMIT, Malout

[2]Student, Dept. of CSE MIMIT, Malout

Abstract

This review paper presents a comprehensive performance analysis of two fundamental pointer-based data structures, the linked list and the skiplist, through the lens of the modern multi-level CPU cache hierarchy (L1, L2, L3). While traditional algorithmic analysis focuses on asymptotic complexity, contemporary hardware realities indicate that memory access patterns and their interaction with the cache often dominate real-world performance. The discussion begins by establishing essential principles of cache architecture and the principle of locality, followed by a survey of seminal works in cache-conscious data structure design that highlight key optimization strategies such as clustering and layout transformation. The comparative analysis exposes the inherent spatial locality limitations of both standard linked lists and skiplists, stemming from their reliance on discontiguous, heap-allocated nodes. Evidence shows that the skiplist's logarithmic search complexity can be undermined by its irregular memory access patterns, frequently producing a higher rate of cache misses than a linear scan over a linked list. The paper concludes by examining modern cache-optimized skiplist variants that blend pointer-based and array-based designs to mitigate these hardware-level penalties, emphasizing that a cache-aware design philosophy is essential for achieving high performance in memory-intensive applications.

Keywords: Data Structures, Skip List, Linked List, CPU Cache, L1 Cache, L2 Cache, L3 Cache, Memory Hierarchy, Perfor- mance Analysis, Cache-Conscious, Spatial Locality, Temporal Locality.

I. Introduction

A. The Processor-Memory Performance Gap

In the landscape of modern computing, a persistent and widening chasm exists between processor speed and main memory latency. This phenomenon, often termed the "Memory Wall," represents a primary bottleneck to application performance [1], [2]. While CPU clock frequencies have scaled exponentially over decades, the access latency of Dynamic Random-Access Memory (DRAM) has improved at a far more modest pace. The consequence is that high-performance processor cores frequently enter stall states, idling for hundreds of clock cycles while waiting for data to be fetched from main memory. This disparity has profound implications for software design, shifting the performance-critical focus from minimizing computational operations to minimizing memory latency.

B. The Multi-Level CPU Cache as a Mitigation Strategy

The fundamental architectural solution to the memory wall is the multi-level CPU cache hierarchy [2], [3]. Modern processors employ a tiered system of small, fast Static RAM (SRAM) caches—typically designated L1, L2, and L3—to serve as intermediaries between the CPU cores and the slower, larger DRAM [4], [5]. By storing frequently or recently accessed data closer to the processor, the cache system effectively hides the high latency of main memory access for a significant portion of memory requests. The efficacy of this system, however, is entirely dependent on the memory access patterns of the running software

C. Pointer-Based Data Structures: A Dichotomy of Design

Pointer-based data structures, such as the linked list and the skiplist, are canonical tools in computer science. Their design elegance and flexibility derive from the use of pointers to connect dynamically allocated nodes, allowing for efficient insertions and deletions without large-scale data movement [6], [7]. This property, known as location transparency, means that the physical memory layout of the structure's elements can be altered without changing the program's semantics [8], [9]. However, this core design feature presents a fundamental conflict with the principles that make caches effective. Standard memory allocators, such as 'malloc', manage a heap of memory with the primary goal of satisfying allocation requests efficiently, not of preserving the logical relationships of a data structure in physical memory [8]. Consequently, logically adjacent nodes in a linked list or skiplist are almost certainly not physically adjacent in memory. This discontiguous allocation pattern leads to pointer-chasing, where traversing the data structure involves a series of memory accesses to disparate, unpredictable locations, a behavior that is inherently hostile to the cache hierarchy [9], [10].

D. Thesis Statement

This paper posits that a purely algorithmic comparison of linked lists (with $O(N)$ search complexity) and skiplists (with $O(\log N)$ expected search complexity) is insufficient and often misleading for predicting performance on modern hardware. A more accurate and insightful analysis must be grounded in the physical realities of the memory hierarchy. The performance of these structures is ultimately governed by the spatial and temporal locality of their memory reference patterns and the resulting cache hit-or-miss profiles. This demonstration shows that the cache-unfriendly pointer chasing inherent in these structures can severely degrade performance, often negating the asymptotic advantages of a more complex structure like the skiplist.

The central thesis is that achieving high performance with such structures requires a cache-conscious design philosophy that explicitly engineers data layouts to align with the operational principles of the CPU cache.

II. Foundational Concepts: Memory Hierarchy and Data Structures

A. The Modern CPU Cache Hierarchy

The CPU cache is a hierarchical system designed to bridge the latency gap between the processor and main memory. It is organized into multiple levels, each with different trade-offs in size, speed, and proximity to the CPU cores [2], [3].

1) **L1 Cache**: The Level 1 (L1) cache is the smallest, fastest, and closest cache to a CPU core. It typically has a latency of only a few clock cycles (e.g., 4 cycles) and is always private to its core [4], [5]. The L1 cache is usually split into two parts: an instruction cache (L1i) for executable code and a data cache (L1d) for program data. Typical sizes for the L1d cache range from 32 KB to 64 KB per core [2]. It serves as the first line of defense against memory latency; an L1 cache hit is nearly as fast as accessing a CPU register.

2) **L2 Cache:** The Level 2 (L2) cache is larger and slightly slower than the L1 cache, with typical latencies around 10-15 clock cycles [4], [5]. L2 caches are often private to each core but can also be shared among a small cluster of cores. Their size ranges from 256 KB to several megabytes [2]. The primary role of the L2 cache is to absorb the majority of L1 cache misses, preventing a more costly access to the L3 cache or main memory.

3) **L3 Cache:** The Level 3 (L3) cache, also known as the Last-Level Cache (LLC), is a much larger cache that is typically shared among all cores on a CPU die [3]. L3 cache sizes can be substantial, ranging from 8 MB to over 100 MB on modern server CPUs [2]. While significantly faster than DRAM, its latency is considerably higher than L1 or L2, often in the range of 40-50 cycles [4]. The L3 cache acts as a final backstop before a memory request must be serviced by main memory, which can have a latency of 300 or more cycles [4].

4) **Cache Lines and Associativity:** Data is transferred between main memory and the cache hierarchy in fixed-size blocks called cache lines, which are typically 64 bytes on modern architectures [11], [12]. When a program requests a single byte from memory, the entire 64-byte cache line

containing that byte is loaded into the cache. This mechanism is crucial for exploiting spatial locality. Set associativity is a design feature that determines where a particular cache line can be placed within the cache, representing a compromise between the simplicity of direct-mapped caches and the flexibility of fully associative caches to reduce conflict misses [12].

B. The Principle of Locality

The effectiveness of the entire cache system is predicated on a property of program behavior known as the principle of locality, which has two primary forms [13], [14].

1) **Temporal Locality:** This principle states that if a program accesses a particular memory location, it is highly likely to access that same location again in the near future [14], [15]. Caches exploit temporal locality by retaining recently accessed data. A common example is a variable used as a loop counter or an accumulator, which is accessed repeatedly in a short time span [15].

2) **Spatial Locality**: This principle states that if a program accesses a memory location, it is highly likely to access nearby memory locations soon after [13], [15]. Caches exploit this by fetching entire cache lines. When data at address X is requested, the data at addresses X + 1, X + 2, . . . , X + 63 is also loaded into the cache, effectively prefetching it [9]. The canonical example of spatial locality is the sequential traversal of an array.

C. Canonical Pointer-Based Structures

The choice of a data structure implicitly defines a program's memory access signature, which in turn determines its interaction with the cache.

1) **The Singly Linked List**: A singly linked list is composed of a series of dynamically allocated nodes. Each node contains a data payload and a pointer to the subsequent node in the sequence. Key operations like insertion and deletion can be performed in O(1) time if the position is known, but searching for an element requires a linear traversal of the list, resulting in an O(N) time complexity [10], [11]. This traversal follows a chain of pointers, 'current
= current->next', where the memory address of the next node is determined by the value stored within the current node. This data-dependent access pattern is fundamentally at odds with spatial locality, as logically adjacent nodes are rarely physically contiguous.

2) **The Skip List**: The skiplist, introduced by William Pugh, is a probabilistic data structure that provides an alternative to balanced binary search trees [16], [17]. It is built upon a sorted singly linked list (level 0) with a hierarchy of additional linked lists that act as ”express lanes.” A node at level i has a pointer to the next node at level i or higher. The level of a new node is determined probabilistically, typically with a 50% chance of being promoted to the next higher level [16]. This structure allows search, insertion, and deletion operations to be performed in expected O(log N) time [11]. A search begins at the highest level, traversing forward as far as possible without overshooting the target, then dropping down to the next level to continue the search. This results in an access pattern characterized by large, unpredictable jumps across memory, which is even more disruptive to spatial locality than the linear scan of a linked list [11], [18].

III. Literature Review on Cache-Conscious Data Structures

The performance degradation caused by the poor locality of pointer-based structures has motivated a significant body of research focused on creating ”cache-conscious” or ”cache-aware” data structures. This field acknowledges that the abstraction of location transparency, while powerful, must be managed to align with hardware realities [6], [8], [9].

A. The Inherent Locality Challenge of Pointer Indirection

B. The core challenge stems from the fact that standard memory allocators are cache-agnostic. Such allocators manage the heap to minimize fragmentation and satisfy allocation requests quickly, without any awareness of an application's data access patterns [8]. This behavior results in a memory layout where logically related data is scattered across physical memory. Seminal work by Chilimbi, Hill, and Larus identified this disconnect as a primary source of performance bottlenecks and demonstrated that reorganizing data layouts to match access patterns can yield substantial performance improvements [6], [8]. This line of research established that careful placement of structure elements is the essential mechanism for improving the cache locality of pointer-manipulating programs. Key Optimization Techniques

The literature proposes several key techniques to transform cache-agnostic layouts into cache-conscious ones. These strategies can be broadly categorized as follows:

1) **Clustering:** Clustering is the technique of co-locating data elements that are likely to be accessed contempo- raneously into the same cache block [9], [19]. This directly improves spatial locality and provides a form of implicit prefetching. When one element in the cluster is accessed, the others are loaded into the cache along with it. For tree-like structures, an effective strategy is subtree clustering, where small subtrees are packed into a single cache block [8]. Another approach is cache-conscious allocation, implemented in tools like ‘ccmalloc‘, where the memory allocator is given a hint (e.g., a pointer to a related node) to guide the placement of a new node near its logical neighbors [19].

2) **Coloring:** Coloring is a technique designed to reduce cache conflict misses. It works by mapping contemporaneously accessed structure elements to non-conflicting regions of the cache [9], [19]. For example, in a tree, the nodes near the root are accessed far more frequently than the leaf nodes. Coloring can segregate these ”hot” root nodes from ”cold” leaf nodes, ensuring that accesses to infrequent nodes do not evict the more critical hot nodes from the cache.

3) **Compression and Field Reordering:** These techniques aim to improve cache line utilization by reducing the memory footprint of data structures. Field reordering involves rearranging the fields within a ‘struct‘ or ‘class‘ definition to group frequently accessed (”hot”) fields together [20]. This increases the probability that all hot fields reside within a single cache line. Cold fields can be moved to the end of the structure or even split off into a separate, indirectly referenced object. This ensures that valuable cache space is not wasted on data that is rarely used [20].

The development of these techniques signifies a crucial evolution in software performance engineering. This shift reflects a move away from treating software and hardware as independent layers of abstraction. The memory wall has effectively broken this abstraction, demonstrating that an O(log N) algorithm with poor locality can be significantly slower than an O(N) algorithm with excellent locality [10]. Modern high-performance data structures, such as the Express Skiplist (ESL) [21], are direct products of this new hardware–software co-design philosophy. The design explicitly hybridizes data layouts—using cache-friendly arrays for some components and flexible pointers for others—to achieve both algorithmic efficiency and hardware affinity. This holistic approach recognizes that peak performance is no longer attainable through software optimization isolated from the hardware on which it runs.

IV. Comparative Analysis of Cache Behavior

A. Linked List Cache Performance Profile

A search operation on a standard linked list provides a clear illustration of the performance penalty of poor spatial locality.

1) Traversal Analysis: Each step in a linked list traversal, ‘current = current->next‘, involves a dependent memory access. The CPU must first load the contents of the ‘current‘ node to retrieve the memory address of the next node. Only then can it issue a memory request for that next address. This creates a serial dependency chain where the latency of each step cannot be hidden by out-of-order execution. Because logically adjacent nodes are physically scattered by the memory allocator, each of these pointer-chasing steps has a high probability of causing a cache miss, forcing the CPU to stall for hundreds of cycles while fetching the next node from L3 or main memory [10].

2) Locality Profile: The spatial locality of a standard linked list is extremely poor. Fetching the 64-byte cache line for one node provides no useful prefetching for the next node in the logical sequence, as it is almost certainly located in a different cache line [6]. Temporal locality is also generally low during a single traversal, as each node is visited only once.

3) Mitigation: Unrolled Linked Lists: A common cache-conscious optimization is the unrolled linked list. This structure modifies the node to contain a small array of elements instead of a single element. Traversal then involves iterating through the array within a node (which has excellent spatial locality) before chasing a pointer to the next node. This reduces the frequency of pointer-chasing and cache misses by a factor equal to the node’s capacity, but it does not eliminate the fundamental problem of cache misses when transitioning between nodes [11].

B. Standard Skip List Cache Performance Profile

While offering superior asymptotic complexity, the standard skiplist's memory access pattern is even more detrimental to cache performance.

1) Search Analysis: A skiplist search traverses pointers at multiple levels, resulting in a series of large and unpredictable jumps in memory. For instance, a search might access a node at address '0x1000', then follow a high-level pointer to a node at '0x9500', then drop to a lower level and access a node at '0x4200'. These erratic memory accesses are the antithesis of spatial locality [11], [18]. This pattern makes it impossible for the CPU's hardware prefetchers, which are designed to detect strided or sequential access patterns, to predict and fetch the next required cache line in advance. Consequently, nearly every node visited during a skiplist search that is not already in the cache will result in a compulsory cache miss with the full latency penalty.

2) Pointer Overhead: A skiplist node must store a variable-sized array of forward pointers, one for each level it participates in. On average, a node in a skiplist with a promotion probability of $p = 0.5$ will have 2 forward pointers [11]. This increases the size of each node compared to a linked list node, reducing the total number of nodes that can fit within a given cache level and potentially increasing the rate of capacity misses.

C. Synthesis and Modern Implementations

The core performance trade-off between a linked list and a skiplist on modern hardware is not simply O(N) versus O(log N). It is a trade-off between the number of node visits and the cache miss penalty per visit. A linked list performs many node visits, each with a high probability of a cache miss. A skiplist performs far fewer node visits, but each visit has an even higher probability of a cache miss due to its more random access pattern. For datasets that are too large to fit in the L3 cache, the cumulative latency of these cache misses can easily overwhelm the algorithmic benefit of the skiplist's logarithmic complexity.

This reality has driven the development of modern, cache-conscious skiplist variants. A prime example is the Express Skiplist (ESL) [21]. The key innovation of ESL is its Cache-Optimized Index Level (COIL). Instead of implementing the upper "express lanes" as pointer-based linked lists, ESL uses arrays of nodes. Searching the index levels thus becomes a series of array lookups, which exhibit excellent spatial locality and are highly amenable to hardware prefetching. This hybrid design leverages arrays for a fast, cache-friendly traversal of the index and retains a traditional linked list at the data level (level 0) to maintain flexibility for insertions and deletions. In evaluations, this cache-aware design was shown to improve throughput by up to 2.8x over other skiplist implementations [21].

The following table provides a synthesized comparison of the cache-relevant characteristics of these data structures.

TABLE I
Comparative Analysis of Data Structure Cache Characteristics

Feature	Standard Linked List	Standard Skip List	Cache-Conscious SL
Asymptotic Search	O(N)	O(log N) expected	O(log N) expected
Memory Layout	Discontiguous nodes	Discontiguous nodes	Hybrid: Array index
Spatial Locality	Very Poor	Extremely Poor	High (index); Poor (data)
Temporal Locality	Low (single traversal)	Low (single traversal)	High (index); Low (data)
L1/L2 Miss Profile	High miss probability	Very high miss prob.	Low (index); High (data)
HW Prefetcher	Ineffective	Ineffective	Highly effective (index)
Primary Bottleneck	Pointer-chasing	Erratic pointer-chasing	Index-to-data transition
Optimization	Unrolling	N/A (inherently non-local)	Hybridization (COIL)

V. Conclusion and Future Directions

A. Recapitulation of Findings

This analysis has demonstrated that for pointer-based data structures like linked lists and skiplists, performance on modern computer architectures is dominated by their cache behavior, not merely their asymptotic complexity. The standard implementations of both structures suffer from poor spatial locality due to their reliance on discontiguous, heap-allocated nodes, leading to frequent and high-latency cache misses. While the skiplist offers a theoretically superior O(log N) search time, its erratic, non-local memory access pattern can result in a higher cache miss penalty per node visit, potentially rendering it slower than a linear scan over a linked list in practice, especially for large, out-of-cache datasets.

B. Concluding Insight

The primary takeaway from this review is that for performance-critical systems, a cache-conscious design is not an optional micro-optimization but a fundamental architectural requirement. The abstraction layers that separate software algorithms from hardware realities are no longer impermeable. High-performance data structures must be designed with an explicit awareness of the memory hierarchy. This often leads to hybrid designs, such as the Express Skiplist, which strategically combine the strengths of different layouts—the cache-friendliness of arrays and the dynamic flexibility of linked lists—to achieve both algorithmic efficiency and hardware affinity.

C. Future Scope

The principles of cache-conscious design remain a fertile ground for research. Several future directions are apparent:

- Persistent Memory: The emergence of non-volatile and persistent memory technologies, which have different latency and bandwidth characteristics than DRAM, will require a rethinking of data structure layouts to optimize for this new tier in the memory hierarchy.
- Adaptive Structures: Research into adaptive data structures that can monitor their own access patterns and dynamically reorganize their physical memory layout—for instance, by switching between a linked-list and an array-based representation—could provide robust performance across diverse workloads.
- Concurrency and NUMA: Extending cache-conscious principles to concurrent data structures on many-core and Non-Uniform Memory Access (NUMA) architectures is critical. In NUMA systems, the cost of accessing remote memory is an order of magnitude higher than local memory, making data locality not just a performance optimization but a prerequisite for scalability.

Ultimately, the path forward in high-performance computing lies in a deeper synthesis of algorithm design and computer architecture, creating software that is not just logically correct but also physically attuned to the hardware it commands.

References

[1] M. Spiegel, ”Cache-Conscious Concurrent Data Structures,” Ph.D. dissertation, University of Rochester, 2012.

[2] Wikipedia contributors, ”Cache hierarchy,” Wikipedia, The Free Encyclopedia, https://en.wikipedia.org/wiki/Cache_hierarchy, (accessed May 20, 2024).

[3] Wikipedia contributors, ”CPU cache,” Wikipedia, The Free Encyclopedia, https://en.wikipedia.org/wiki/CPU_cache, (accessed May 20, 2024).

[4] ”L1, L2, L3 Cache Explainer,” Reddit discussion, r/hardware, 2023. Available: https://www.reddit.com/r/hardware/comments/ 1hr5o8a/l1_l2_l3_explainer/

[5] M. Anderson, ”The Cache Clash: L1, L2, and L3 in CPUs,” Medium, 2023. Available: https://medium.com/@mike.anderson007/ the-cache-clash-l1-l2-and-l3-in-cpus-2a21d61a0c6b

[6] T. M. Chilimbi, M. D. Hill, and J. R. Larus, ”Cache-conscious structure layout,” in Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation (PLDI ’99), 1999, pp. 1–1

[7] Wikipedia contributors, ”Pointer (computer programming),” Wikipedia, The Free Encyclopedia, https://en.wikipedia.org/wiki/ Pointer_(computer_programming), (accessed May 20, 2024).

[8] T. M. Chilimbi, M. D. Hill, and J. R. Larus, ”Cache-Conscious Structure Layout,” Technical

Report, University of Wisconsin-Madison, 1999.

[9] T. M. Chilimbi, M. D. Hill, and J. R. Larus, ”Cache-conscious data structures: design and implementation,” IEEE Computer, vol. 33, no. 12, pp. 67-74, 2000.

[10] B. Stroustrup, ”Why you should avoid linked lists,” CppCon 2014 talk discussion. Available: https://www.reddit.com/r/programming/ comments/260vg6/after_watching_bjarne_stroustrup_why_you_should/

[11] Stack Overflow community, ”Realistic usage of unrolled skip lists,” Stack Overflow, 2015. Available: https://stackoverflow.com/ questions/28539701/realistic-usage-of-unrolled-skip-lists

[12] ”Memory Hierarchy,” LibreTexts Engineering, 2021. Available: https://eng.libretexts.org/Courses/Delta_College/Introduction_to_ Operating_Systems/05%3A_Computer_Architecture_-_Memory/5.01%3A_Memory_Hierarchy

[13] Wikipedia contributors, ”Locality of reference,” Wikipedia, The Free Encyclopedia, https://en.wikipedia.org/wiki/Locality_of_ reference, (accessed May 20, 2024).

[14] ”Difference between Spatial Locality and Temporal Locality,” GeeksforGeeks, 2023. Available: https://www.geeksforgeeks.org/ computer-organization-architecture/difference-between-spatial-locality-and-temporal-locality/

[15] S. Lumetta, ”Caches and Locality,” CSE378 Lecture Notes, University of Washington, 2010.

[16] W. Pugh, ”Skip lists: A probabilistic alternative to balanced trees,” Communications of the ACM, vol. 33, no. 6, pp. 668–676, 1990.

[17] W. Pugh, ”Skip Lists: A Probabilistic Alternative to Balanced Trees,” Technical Report CS-TR-2196, University of Maryland, 1989.

[18] Stack Overflow community, ”Why skiplist memory locality is poor but balanced tree is good,” Stack Overflow, 2016. Available: https://stackoverflow.com/questions/35647064/why-skiplist-memory-locality-is-poor-but-balanced-tree-is-good

[19] T. M. Chilimbi, M. D. Hill, and J. R. Larus, ”Cache-Conscious Structure Layout,” in Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Atlanta, GA, USA, May 1999, pp. 1-12.

[20] T. M. Chilimbi, B. Davidson, and J. R. Larus, ”Cache-conscious structure definition,” in Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation (PLDI ’99), 1999, pp. 13–24.

[21] Y. Na, B. Koo, T. Park, J. Park, and W. H. Kim, ”ESL: A High-Performance Skiplist with Express Lane,” Applied Sciences, vol. 13, no. 17, p. 9925,