The Storage Hierarchy is Not a Hierarchy: Optimizing Caching on Modern Storage Devices with Orthus

Kan Wu, Zhihan Guo, Guanzhou Hu, Kaiwei Tu, Ramnatthan Alagappan[†],
Rathijit Sen[‡], Kwanghyun Park[‡], Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
University of Wisconsin–Madison [†]VMware Research [‡]Microsoft

1 Introduction

The notion of hierarchy pervades computer systems. For instance, memory and storage systems are organized in a hierarchy, with each layer providing higher capacity but lower performance than the layer above it.

A long-standing strategy to manage such hierarchies is that of *caching*. Because the fast layer (D_{hi}) usually offers significantly higher performance than the capacity layer (D_{lo}), caching strives to ensure that most accesses hit the fast layer by placing frequently-accessed data in it. Over the years, caching has thus been optimized for one goal: maximizing the hits to the fast layer.

While this goal may be appropriate for traditional hierarchies, it is ill-suited and inadequate for hierarchies with emerging storage devices. The key problem is that modern devices exhibit overlapping performance characteristics. For example, as shown in Figure 1, in some situations (namely, with low levels of parallelism), the performance device outperforms the capacity one substantially (like in a traditional hierarchy), and thus classic caching would work well. However, in some circumstances (e.g., with high levels of parallelism), the devices deliver similar performance. In some cases, the "slow" device may even offer better performance than the "fast" device (e.g., NVM vs Optane with many concurrent writes). Classic caching, which focuses solely on maximizing hit rate, cannot reap the significant performance offered by the capacity device. Thus, it is essential to rethink how to manage modern devices in the hierarchy.

We present *non-hierarchical caching* (NHC) [2][1], a new approach to caching in modern storage hierarchies. The key insight behind NHC is to route some requests to the capacity device, in cases where sending them to the fast device does not yield higher performance. The result is that NHC delivers the aggregate performance of *all* the devices in the hierarchy (in contrast to classic caching that can approximate the performance of only the fast device).

We have implemented NHC in a block-layer caching kernel module and a user-space caching layer for a key-value store. Our experiments show that NHC delivers substantially better performance (by up to 2×) than classic caching on several modern hierarchies under a range of realistic workloads.

2 Non-Hierarchical Caching

NHC uses a new architecture and a novel cache-scheduling algorithm to effectively use the performance of the capacity device. NHC is compatible with all classic caching policies, performs strictly better than classic caching, and needs no prior knowledge of the devices or the workload.

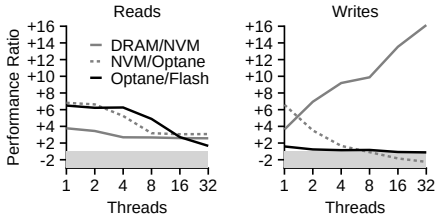

Figure 1: **Performance Ratios Across Modern Devices.** *The figure compares 4KB reads/writes throughput across device pairings (NVM: an Intel Optane DCPM module, Optane: an Optane 905P SSD, Flash: a Flash SSD). For any pair X/Y, $\frac{X}{Y}$ is plotted if the performance of $X \geq Y$; otherwise, $\frac{-Y}{X}$ is plotted(in the gray region). Note there is no value between -1 and +1.*

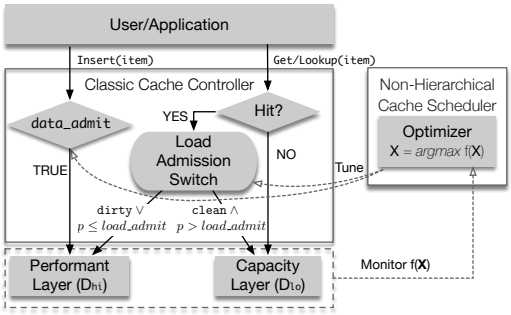

Figure 2: **Non-Hierarchical Cache Architecture.** *NHC is transparent to users as before. Every classic caching implementation can be converted to a NHC one. Notice that the decision points only tune read hits/misses.*

2.1 NHC Architecture

Classic caching can be upgraded to NHC by introducing a set of decision points to its cache controller and a *cache scheduler* as in Figure 2. The classic cache controller is used to direct application reads and writes to devices (e.g., D_{hi} and D_{lo}), as well as to control content in D_{hi} (e.g., according to LRU). The *non-hierarchical cache scheduler* tracks performance and controls the behavior of cache controller.

A boolean flag `data_admit (da)` and a variable `load_admit (la)` enable the NHC scheduler to perform this control. When a **read miss** occurs on D_{hi}, the `da` flag determines the actions: when `da` is set, the missed data item is handled by the classic cache-replacement policy; when `da` is not set, D_{lo} directly serves the data. Classic caching is the case where `da` is true.

The `la` variable regulates how **read hits** are treated and indicates the percentage of read hits to be sent to D_{hi}; if `la` is 0, every read hit is sent to D_{lo}. Specifically, a random number

[1] `research.cs.wisc.edu/adsl/Publications/fast21-kan.pdf`

$R \in [0, 1.0]$ is generated upon a read hit; the request is sent to D_{hi} when $R <= \texttt{la}$; otherwise, it is sent to D_{lo}. `la` is always 1 in classic caching.

NHC is compliant with any classic write-allocation policy that manages **write hits/misses**. According to the policy, NHC decides whether to allow write misses into D_{hi} or not; `da` and `la` do not influence writes. If the system is set up with write-back, NHC does not send dirty read hits to D_{lo}.

2.2 NHC Scheduler Algorithm

The NHC scheduler adjusts the behavior of the cache controller to optimize a target metric. The target may either be a user-level (e.g., ops/sec) or device-level (e.g., request latency) metric. f is a function that measures the target metric. The scheduler has two states: to increase D_{hi}'s hit rate by performing classic caching, or to maintain the cached data constant while adjusting the load sent to each device.

State 1: Improve hit rate. In this state, NHC acts the same as classic caching (with `da` as true and `la`$= 1$). By doing so, NHC warms up D_{hi} with hot data items, increasing the hit rate for D_{hi}. The NHC Scheduler tracks the hit rate and ends this phase when the hit rate is relatively stable.

State 2: Adjust load between devices. After D_{hi} has a high hit rate, the NHC scheduler will examine whether offloading some read hits to D_{lo} would improve the overall performance. `da` is set to false and feedback (Δf) is used in this state to optimize the performance target. `la` is iteratively calibrated to improve the target metric. In every iteration, like gradient-descent, $f(\texttt{la} \pm \text{step})$ is measured; `la` is then updated in the direction to enhance f. If the scheduler finds that optimal `la` is 1, it ceases tuning and goes back to State 1. When the workload locality varies substantially, the NHC scheduling procedure is restarted.

3 Evaluation

We have implemented NHC in *Orthus-CAS*, a block-layer caching kernel module, and *Orthus-KV*, a user-level caching layer for a key-value store. Our implementations support target functions such as throughput (default), and average and tail (P99) latency.

3.1 Orthus-CAS on Various Storage Hierarchies

Figure 3 shows the performance of Orthus-CAS in multiple modern hierarchies. Orthus-CAS works in the same manner as classic caching when the load is light. However, when the workload is able to exploit the performance device, NHC substantially outperforms classic caching by leveraging the usable performance from the capacity device. Under high load (e.g., Load-2.0), NHC increases performance by 21% - 54% for DRAM+NVM, NVM+Optane and Optane+Flash hierarchies. Our tests with FlashSim also show how practitioners can predict the benefit of using NHC in their hierarchies.

3.2 Orthus-KV with Dynamic Workloads

Figure 4 demonstrates how NHC manages shifts in workload with the Facebook ZippyDB benchmark [1]. As shown in the top graph, Orthus-KV beats classic caching by up to 100% during the day. The bottom graph illustrates how Orthus-KV changes data and load admit ratios. They are both around 100% throughout the night. During the day, Orthus-KV holds the data admit ratio at 0 and changes the load admit ratio to respond to varying loads.

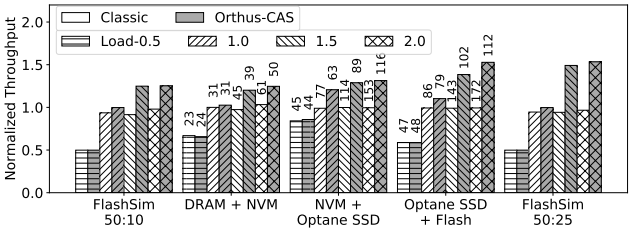

Figure 3: **Orthus-CAS on Various Hierarchies.** *The figure shows NHC's performance under read-only workloads (95% hit rates) with different amounts of load. We use the same devices as in Figure 1. We also use FlashSim to simulate devices with different performance characteristics. Load-1.0 is specified as the minimum load to reach full cache-device read bandwidth; Load-0.5, 1.5, and 2.0 are created by scaling Load-1.0. All throughputs are normalized to the maximum read bandwidth of the caching device. Latency (μs) is shown above each bar (not comparable across hierarchies).*

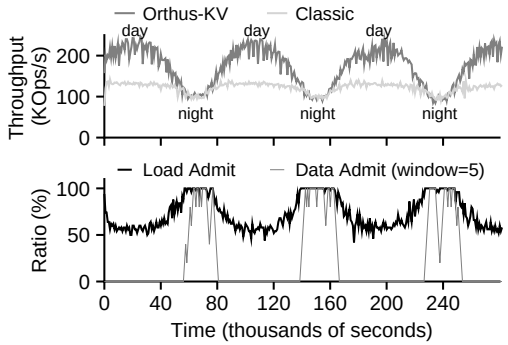

Figure 4: **Orthus-KV With ZippyDB workloads.** *This figure shows the Orthus-KV throughput (top) as well as load/data admit ratios over time in Orthus-KV (bottom). We focus on the hierarchy of Optane/Flash. The Facebook ZippyDB benchmark generates key-value operations based on realistic trace statistics; the request rate models the diurnal load sent to ZippyDB servers. We speed up the replay by 1000 to stress the storage system and to better evaluate NHC's reactions to shifting loads.*

Acknowledgments

We thank NSF (CNS-1838733, CNS-1763810), VMware, Intel, Seagate, and Microsoft for their support. Opinions expressed in this material are those of the authors.

References

[1] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. Characterizing, modeling, and benchmarking RocksDB key-value workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 209–223, 2020.

[2] Kan Wu, Zhihan Guo, Guanzhou Hu, Kaiwei Tu, Ramnatthan Alagappan, Rathijit Sen, Kwanghyun Park, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. The Storage Hierarchy is Not a Hierarchy: Optimizing Caching on Modern Storage Devices with Orthus. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 307–323, 2021.