Communicating Research to the General Public

The WISL Award for Communicating PhD Research to the Public launched in 2010, and since then over 100 Ph.D. degree recipients have successfully included a chapter in their Ph.D. thesis communicating their research to non-specialists. The goal is to explain the candidate's scholarly research and its significance—as well as their excitement for and journey through their area of study—to a wider audience that includes family members, friends, civic groups, newspaper reporters, program officers at appropriate funding agencies, state legislators, and members of the U.S. Congress.

WISL encourages the inclusion of such chapters in all Ph.D. theses everywhere, through the cooperation of PhD candidates, their mentors, and departments. WISL offers awards of $250 for UW-Madison Ph.D. candidates in science and engineering. Candidates from other institutions may participate, but are not eligible for the cash award. WISL strongly encourages other institutions to launch similar programs.

Wisconsin Initiative for Science Literacy

The dual mission of the Wisconsin Initiative for Science Literacy is to promote literacy in science, mathematics and technology among the general public and to attract future generations to careers in research, teaching and public service.

Contact: Prof. Bassam Z. Shakhashiri

UW-Madison Department of Chemistry

bassam@chem.wisc.edu

www.scifun.org

Layout-Aware Data Organization For Heterogeneous Hierarchies

by

Vinay S. Banakar

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2026

Date of final oral examination: Feb 20th, 2026

The dissertation is approved by the following members of the Final Oral Committee:

Andrea C. Arpaci-Dusseau, Professor, Computer Sciences
Remzi H. Arpaci-Dusseau, Professor, Computer Sciences
Kimberly Keeton, Principal Software Engineer, Google
Michael M. Swift, Professor, Computer Sciences
Shivaram Venkataraman, Associate Professor, Computer Sciences
Dimitris Papailiopoulos, Associate Professor, Electrical Engineering

Chapter 8

Warehouses Inside Your Computer

In this chapter, I present my research in a form geared towards a general public audience. I hope to make the results of our work more accessible. Access to knowledge is a powerful tool in developing an engaged and compassionate society, and we are proud to contribute in our own small way. Many thanks to the Wisconsin Initiative for Science Literacy (WISL), including Bassam Shakhashiri, Cayce Osborne, and Elizabeth Reynolds, for providing this opportunity to communicate my work to a broader audience.

8.1 The Warehouse Problem

One number changed how I thought about computers. While analyzing data from Google's servers, I discovered that for every dollar spent on fast memory, as little as four cents was going toward data anyone actually used. The other ninety-six cents paid for dead weight: data sitting in expensive storage that no program had touched in hours, days, sometimes ever. I had expected some waste. I had not expected that kind of waste.

To understand how this happens, think about a warehouse. You tap "buy" on your phone, and a package shows up the next day. That speed is possible because the warehouse stored products in the right places. Bestselling items sit on shelves right next to the shipping dock, ready to be grabbed and loaded onto a truck in seconds. Rarely ordered items live in back storage,

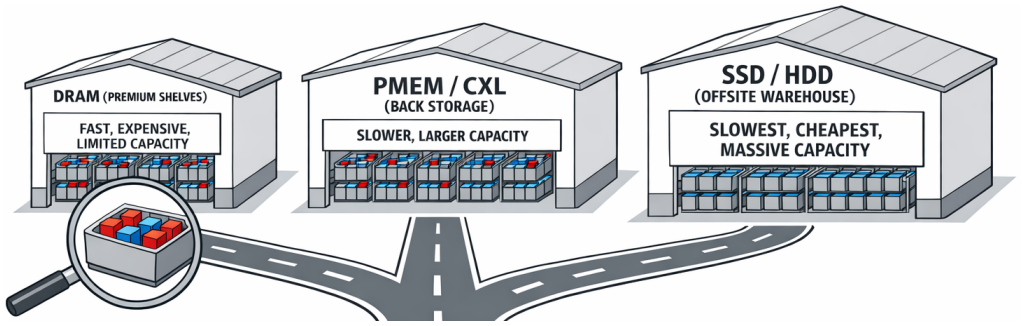

Figure 8.1: **The warehouse problem.** Your computer's memory works like a chain of warehouses, from a premium facility next to the shipping dock (fast, expensive, limited) to back storage and offsite warehouses (slower, cheaper, larger). The operating system moves data between them in fixed-size bins called *pages*. The magnified bin in the lower left shows the core problem: because the manager can only move whole bins, a single popular product (red) traps an entire bin full of unpopular products (blue) in premium space, wasting expensive shelf real estate on data nobody is using.

where shelf space is cheaper and nobody minds the longer walk.

This simple principle, put popular things where they are easy to reach, seems obvious. But what happens when the warehouse is disorganized? Imagine that products are shelved based on *when* they arrived, not *how often* they are ordered. A best-selling phone case ends up in the same bin as a novelty item nobody wants. The warehouse manager can only move *entire bins*, never individual products inside them. So a single popular product traps its entire bin in premium space near the dock, wasting expensive shelf real estate on products nobody is ordering. Meanwhile, popular items stuck in back-storage bins take ages to retrieve (Figure 8.1).

This is exactly the problem your computer's memory faces. Modern computers do not have just one kind of memory. They have a hierarchy, ranging from small, fast, and expensive memory at the top to large, slow, and cheap memory at the bottom. The operating system, which manages all of this memory, works like the warehouse manager: it can move data around, but only in fixed-size chunks called *pages*. It cannot reach inside a page to

move one piece of data without moving everything else on that page. The premium shelves correspond to your computer's fast memory (a technology called DRAM), and back storage corresponds to slower, cheaper memory technologies. Each bin is a memory page, and each product is a piece of data.

These different kinds of memory do not merely differ in speed. They differ in character. Some can hand you a single byte; others insist on giving you a whole block whether you want it or not. Some read quickly, but write data slowly. Some slow down when reading and writing happen at the same time. An approach that works beautifully on one kind of storage can perform terribly on another, the way a shelving strategy designed for a walk-in closet falls apart in a three-story warehouse. Any serious attempt to organize data must account for the specific properties of the hardware it lives on.

When popular data, data your programs touch all the time, gets mixed with unpopular data on the same pages, the result is waste on a staggering scale. This is the problem I mentioned at the start. Across real workloads at companies like Google and Meta, I found that for every 100 megabytes of memory the operating system considers "active," up to 96 megabytes is actually cold data that nobody is using. It sits trapped in expensive fast memory by a few hot neighbors, just like that novelty gift trapping its whole bin next to the shipping dock.

My dissertation asks a simple question: ***can we reorganize how data is laid out in memory so that the operating system's fixed-size view of the world actually works well?*** The answer, it turns out, is yes. But the right approach depends on what you know about the data or how it will be accessed.

Sometimes you know in advance which data will be popular and which will not. A sorting algorithm, for instance, compares small labels (called *keys*) over and over again but only touches the full records (called *values*) at the very end. In that case, you can plan ahead: keep the labels close at hand and leave the bulky records in cheap storage until you need them. But planning ahead also means understanding the hardware you are planning

for. A new kind of computer storage arrived in recent years with a set of properties no previous device had combined: it could read tiny amounts of data from random locations almost as fast as reading sequentially, but writes were far slower than reads, and reading and writing at the same time caused both to degrade. I built *WiscSort* [1], a sorting system designed around these specific properties. It separates the lightweight labels from the bulky products and sorts only the labels. Then it exploits the fast random reads of modern storage to fetch the products at the very end. It also carefully schedules reads and writes so they stay out of each other's way. The result is two to three times faster than the best prior approaches.

Other times, you have no idea what will be popular. A social media platform cannot predict which posts will go viral. A database does not know which records a user will search for next. In these cases, the only option is to watch and reorganize: monitor which data is being accessed, and periodically move popular data to fast memory and unpopular data to slow memory. Think of a warehouse crew that tracks which products are selling and re-sorts the shelves every night.

In a physical warehouse, this reorganization is straightforward. You pick up a box, carry it to a new shelf, and set it down. If a coworker asks where the box went, you tell them. In a computer, the situation is fundamentally harder. Dozens of programs may be reading a piece of data at the same instant, and each of them holds an address, a note that says "this data lives at location 7042." If you move the data to location 2058 while a program is in the middle of reading location 7042, that program gets nonsense. If you freeze every program while you move the data, you destroy performance. The reorganization must happen while the warehouse is open and busy, without anyone ever following an address to the wrong place. I built a system called *OBASE* [2, 3] that solves this problem. It observes which data is hot and reorganizes the layout continuously. At the old address of every moved object, it installs a forwarding label, so any program still holding the

old address is silently redirected to the new one. No program ever pauses, and no program ever finds stale data. The result is up to 70% less wasted memory, with almost no slowdown.

In the rest of this chapter, I will walk you through how your computer's memory actually works, why the current approach wastes so much of it, and how WiscSort and OBASE fix the problem. All you need is the warehouse in your mind.

8.2 Why Bins?

If mixing popular and unpopular products together causes so much waste, the obvious question is: why not track individual products instead of bins? If the warehouse manager knew the exact shelf location of every single product, she could move just the popular ones to premium shelves and leave everything else in back storage. No bins, no wasted space, no problem.

The answer is the same in a warehouse and in a computer: keeping track of everything individually is too expensive. A large warehouse might stock tens of millions of products of different sizes. A master catalog listing the exact shelf location of every single one would fill an entire room, and a worker would spend more time flipping through the catalog than actually retrieving products. The solution is the same one librarians have used for centuries: group items into sections, and keep a much smaller catalog that tracks sections rather than individual books.

In a computer, this smaller catalog is a piece of hardware called the *translation lookaside buffer*, or TLB for short. You can think of it as a small index-card box bolted to the front desk of the warehouse. Every time a worker (the processor) needs a piece of data, it first checks the catalog to find which bin the data lives in and where that bin sits on the shelves. If the bin is listed, the worker gets an answer almost instantly, in about a billionth of a second. If not, the worker has to walk to the back office and search through

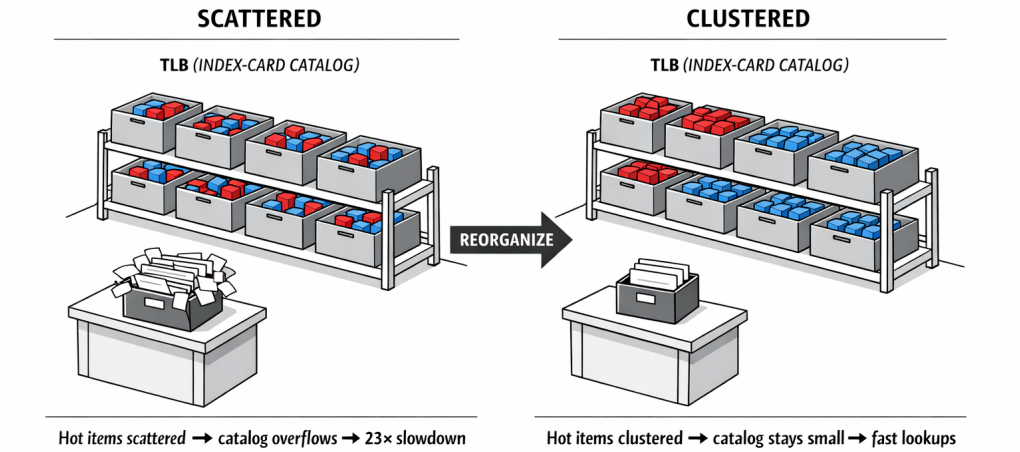

Figure 8.2: **The catalog problem.** When popular products (red) are scattered across many bins, the index-card catalog must track every bin and overflows. When popular products are clustered into just a few bins, the same catalog holds everything it needs with room to spare. The data and the work are identical in both panels; only the *layout* has changed.

a much larger set of records. That detour is about a hundred times slower.

The catalog can only hold a limited number of entries, typically a few thousand. When the data a program is actively using fits into a small number of bins, the catalog works beautifully: it holds every bin the worker might need, and lookups are nearly free. But when popular data is scattered across many bins, each containing just a few hot items mixed with cold ones, the catalog overflows. The worker constantly looks up a bin, finds it missing, walks to the back office, retrieves the location, evicts an old entry from the full card box, and repeats. In one of our analysis, this kind of catalog overflow caused a program to run *twenty-three times slower* than it did when the same data was clustered into fewer bins. The program was doing the exact same work on the exact same data. The only difference was where the data had been placed.

This is the hidden cost of the bin system. Bins make the catalog small enough to be practical, but they also create a rigid constraint: the operating

system sees bins, not products. If popular products are scattered across many bins, two bad things happen at once. First, the premium shelves fill up with bins that are mostly empty space, the memory waste I discussed in the last section. Second, the catalog overflows, and the processor grinds to a halt looking up bin locations. The two problems reinforce each other, and both share the same root cause: popular and unpopular data mixed together on the same pages.

This also means that any system that clusters popular data into fewer bins earns a double benefit. It frees up premium shelf space *and* it shrinks the catalog back to a manageable size. Both WiscSort and OBASE do exactly this, and both reap the rewards. I will show how in the next two sections.

8.3 Not All Warehouses Are the Same

In Sec. 8.1, I described a sorting algorithm that separates lightweight labels from bulky products and sorts only the labels. That is the core idea behind WiscSort. But to understand *why* it works so well, you need to understand the hardware it was designed for, because that hardware breaks nearly every assumption older sorting systems relied on.

For decades, computer storage meant hard drives: spinning metal platters with a tiny mechanical arm that reads data by physically sliding across the surface, the way a record player's needle traces a groove. Reading data in order, one piece right after the next, was fast because the arm barely had to move. But jumping to a random location meant the arm had to swing across the platter, wait for it to spin to the right spot, and only then start reading. That random jump was roughly a *hundred thousand times* slower than simply reading the next piece in sequence. Because of this, every sorting algorithm built for hard drives followed the same basic strategy: always read and write sequentially, and never, ever jump around.

Then a new kind of storage arrived, one that blurred the line between

memory and disk. Intel's Optane was the most prominent example. In warehouse terms, imagine a new storage facility where the layout has changed in five unexpected ways. First, you can grab a single product off a shelf without pulling the entire bin (*byte addressability*). Second, sending a worker to grab one item from a random shelf is nearly as fast as having them walk down the aisle picking up items in order (*fast random reads*). Third, putting products *onto* shelves is much slower than taking them off; reads are about three times faster than writes (*asymmetric costs*). Fourth, if workers are simultaneously loading shelves and unloading them, they get in each other's way and both slow down (*read-write interference*). Fifth, adding more workers helps only up to a point, especially for loading; beyond that, they start bumping into each other (*constrained concurrency*).

No previous device had combined all five of these properties. Hard drives were slow at everything but predictable. Solid-state drives were faster but still insisted on handing you a whole block at a time. This new storage was fast and flexible in some ways, painfully constrained in others. The key insight is that designing for one or two of these properties is not enough. A system that exploits fast random reads but ignores the interference between reads and writes will sabotage itself. WiscSort was designed around all five simultaneously. (In our technical work, I named this combination of properties the *BRAID* model, after the initials of each property, but the name matters less than the idea: you have to respect the whole package.)

Here is how it works in practice (Figure 8.3). Suppose you have a dataset of one hundred million records, each consisting of a 10-byte label and a 90-byte product. A traditional sorting algorithm reads the full 100 bytes of every record, shuffles them around repeatedly during sorting, and writes them all back. WiscSort reads only the 10-byte labels, each paired with a tiny pointer noting where the corresponding product lives in storage. It sorts these label-pointer pairs entirely in fast memory. Only at the very end does it use the device's fast random reads to fetch each product from its

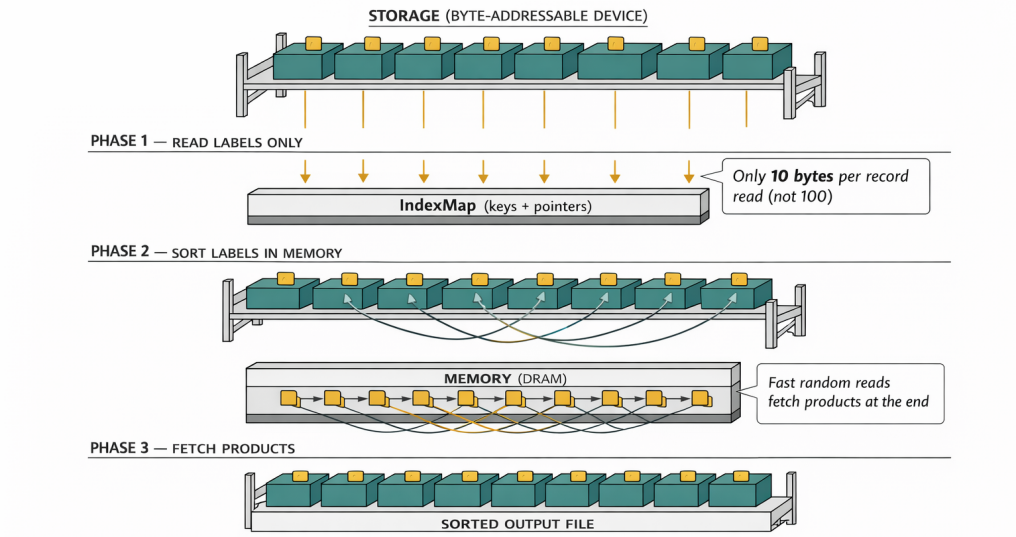

Figure 8.3: **How WiscSort works.** Traditional sorting carries each product (value) alongside its shipping label (key) through every stage. WiscSort separates the two. In the first phase, it reads only the lightweight labels from storage using fine-grained byte accesses, and sorts them in memory. In the final phase, it uses the fast random reads of modern storage to fetch the corresponding products and assemble the sorted output. Because labels are far smaller than products (often one-tenth the size), this dramatically reduces the data that must be read and written.

original location and write the final sorted result.

The difference in data movement is stark. For those hundred million records, traditional sorting reads and writes around 19 gigabytes of data, most of it products being carried along for the ride. WiscSort reads only 1.9 gigabytes of labels during sorting and fetches the products in one final pass. That is a tenfold reduction in intermediate traffic. Because writing is the slowest operation on this hardware, moving less data translates directly into faster completion.

But reducing traffic is only half the story. WiscSort also manages *how* it accesses the device. At startup, it runs a brief calibration step to measure the device's actual performance: how many workers can read at once before

throughput plateaus, how many can write, and when interference kicks in. Based on these measurements, WiscSort assigns separate pools of workers for reading and writing, each sized to match the device’s sweet spot. Crucially, it never lets reading and writing happen at the same time. When data is being loaded from storage, no writing occurs; when sorted data is being written back, all reading pauses. This is the interference-aware scheduling that the intro alluded to, and it prevents the mutual degradation that costs other systems up to fifty percent of their throughput.

I tested WiscSort on the industry-standard sorting benchmark, which stress-tests storage systems by sorting records with small labels and large values. WiscSort’s one-pass version finished three times faster than a competitive traditional approach. Even the two-pass version, needed when the dataset is too large to sort in one shot, was twice as fast. Compared to the best prior system designed for this kind of hardware, WiscSort was seven times faster. The gap comes from treating the device’s five properties as a unified design constraint rather than addressing them one at a time.

8.4 Reorganizing While the Warehouse Is Open

WiscSort works because the task itself reveals which data is hot and which is cold: labels are always hot, products are cold until the end. But most software does not come with that kind of road map. A key-value store (a system that looks up records by name, like a phone book) serving a social media feed, a caching layer at a major web company, a database answering search queries: in all of these, any piece of data might suddenly become popular or fall out of use, and the pattern shifts constantly. I looked at real usage logs from Meta and Twitter and found that for most items, the time between one use and the next swings a lot; over five times for three-quarters of items, and over thirty times for nearly two-thirds. Nothing about when an item was created or where it was allocated tells you how popular it will be next week.

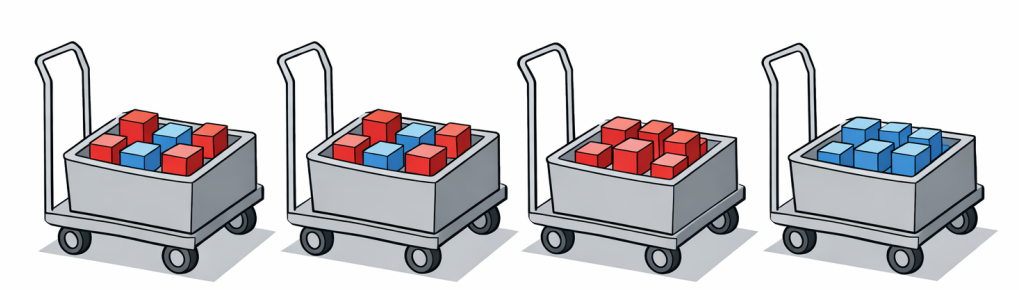

Figure 8.4: **Before and after OBASE.** The left two bins show the typical state of memory: popular products (red) and unpopular products (blue) mixed together on every page. Because the warehouse manager can only move whole bins, every bin must stay in premium space as long as it contains even one popular item. The right two bins show the same data after OBASE has reorganized it: popular products are clustered into one bin, unpopular products into another. Now the cold bin can be safely moved to back storage, freeing premium shelf space without losing any popular items.

When you cannot predict popularity, you must observe it. OBASE does this by continuously watching which data is being accessed, classifying it as hot or cold, and physically reorganizing the layout so that hot objects cluster onto the same pages and cold objects cluster onto others (Figure 8.4). The warehouse manager, who can still only move whole bins, now finds that cold bins are *genuinely* cold: every product inside is unused, so the entire bin can be safely moved to back storage. Hot bins are densely packed with popular products, making the premium shelves far more productive.

This sounds straightforward in principle. In practice, it requires solving four problems that have no easy parallels in a physical warehouse.

Forwarding labels. The first problem is that the most widely used programming languages for building fast, performance-critical software (languages called C and C++) give programmers direct control over where data lives in memory. That power comes at a cost: every part of the program that uses a piece of data holds a note saying "this lives at address 7042." If

you move the data to address 2058, every one of those notes becomes wrong, and there is no built-in mechanism to fix them. Some newer programming languages (like Java or Python) trade away that direct control in exchange for a safety net. They insert a manager between the program and memory, and when data moves, the manager quietly updates all the notes for you. C and C++ have no such manager. The programmer is talking directly to the hardware, which is exactly what makes these languages fast, and exactly what makes reorganizing data so hard.

OBASE introduces a lightweight intermediary I call a *guide*. Instead of pointing directly at the data, every reference points at the guide, and the guide holds the data's current address. When the data moves, only the guide needs to be updated: one single value in one place. Every part of the program that follows the guide is silently redirected to the new location. The guide is the forwarding label from our warehouse analogy: a small card left at the old shelf location that says "this product is now on shelf B-14." Any worker who arrives at the old location reads the card and walks to the new one without even realizing the product was moved.

The cost of this indirection is small. Each time a program accesses data through a guide, it performs one extra lookup, around five billionths of a second, comparable to reading a value already sitting in the processor's fastest cache. For most programs, this adds between two and five percent to the total running time.

Tracking popularity. The second problem is knowing which data is hot and which is cold. OBASE cannot afford to maintain a separate log of every access; at the rate modern processors touch data, such a log would itself become a bottleneck. Instead, OBASE hides a tiny tracking flag inside the guide itself. Here is the trick: every guide contains an address, a string of digits that tells the processor where to find the data. But the address is longer than it needs to be, the way a phone number might have unused

digits at the end. OBASE repurposes a few of those spare digits to record whether the data has been accessed recently. Every time a guide is followed, one of those spare digits is flipped. No separate data structure, no external log, no additional memory. The tracking rides along with the access itself, like a turnstile counter built into the warehouse doorframe that clicks each time a worker walks through.

A background process I call the *Object Collector* periodically reads these flags, about once every two minutes by default, and classifies each object as hot or cold based on recent activity. Objects that have not been accessed for several consecutive windows are marked cold. Objects accessed in any window are marked hot. The threshold for "cold enough to move" adjusts automatically: if the system finds that it is moving objects that soon get accessed again, it becomes more conservative; if almost nothing it moves is ever revisited, it becomes more aggressive. This self-tuning means OBASE adapts to each workload without manual configuration.

Three zones. Based on these classifications, OBASE maintains three zones in memory: NEW, for freshly created data whose popularity is not yet clear; HOT, for the current working set; and COLD, for data that has been idle long enough to be considered inactive. Each zone occupies a contiguous stretch of memory, so the operating system sees clean, uniform regions. A cold zone is a cold zone: every object inside is genuinely idle, and the entire region can be moved to slower memory or reclaimed with confidence. This is the key difference from the status quo, where the operating system looks at a page and sees a mix of hot and cold data, unable to act without risking the hot items.

Data flows between zones as its popularity changes. A freshly created object starts in NEW. If the Object Collector sees it being accessed, it migrates to HOT. If it sits idle for several windows, it migrates to COLD. And if a cold object becomes popular again, it moves back to HOT. The layout is never static; it continuously tracks the application's shifting working set.

Moving products without closing the warehouse. The hardest challenge is the last one: how do you move data while programs are actively reading it? Pausing every program while you move an object, the way a garbage collector might in Java, is unacceptable in the performance-critical systems OBASE targets. But moving data out from under an active reader could corrupt the program's state.

OBASE uses a "try and check" approach (database researchers call it *optimistic concurrency control*, but the intuition is simple). Assume the move will succeed, do the work, and verify at the end that nothing went wrong. Concretely, the Object Collector copies the data to its new location in the target zone. Then it attempts to update the guide in a single all-or-nothing operation: either the update succeeds completely, or it fails completely, with no in-between state. If any program accessed the data while the copy was happening, the guide will have changed, the update fails, and the move is simply abandoned. The data stays where it was, perfectly intact. The program that was reading it never noticed a thing.

This "try and check" approach has a beautiful property: programs never wait on the reorganization crew, and the reorganization crew never corrupts data. Frequently accessed objects naturally resist being moved, because someone is always reading them, and their guide changes before the move can commit. Cold objects, by definition, are not being read, so their moves almost always succeed. The system migrates exactly the objects that can safely be migrated, without anyone having to coordinate.

Letting the manager do her job. One final design choice deserves emphasis, because it is the reason OBASE works with so many different systems. OBASE does not decide what to do with cold data. It does not page it to disk, compress it, or send it to slower memory. All it does is reorganize the address space so that cold data and hot data are no longer mixed together. The existing warehouse manager, whether that is Linux's built-in memory

reclaimer, Meta's TMO system, or a hardware-aware tiering engine like Memtis, continues to make the actual decisions about which bins to move where. But now, when the manager looks at a bin and asks *is everything in here cold?*, the answer is reliably yes.

This separation is what makes OBASE practical to deploy. A large company does not need to replace its existing memory management infrastructure. It plugs OBASE in as a frontend that improves the layout, and every backend it already uses becomes more effective. I tested OBASE with six different memory management systems and every single one improved. Some improved dramatically. Linux's built-in reclaimer, without OBASE, can reduce memory usage from 13 gigabytes to about 7 by reclaiming only the pages it is confident are cold. With OBASE preparing the layout, the same reclaimer reaches 4 gigabytes, the theoretical minimum, with no loss in performance. The reclaimer did not get smarter. It just finally had accurate information to work with.

In tiered memory configurations, where a computer has a small amount of fast memory and a larger pool of slower memory, OBASE is equally effective. Without OBASE, the popular data is spread across so many pages that it cannot fit in the fast tier, even when the actual amount of popular data is small. With OBASE compacting the popular data into fewer pages, the fast tier has room to spare. In our experiments, OBASE allowed the system to achieve the same performance with *half* the fast memory, effectively doubling the capacity of the expensive tier for free. Across all workloads and configurations, the overhead of running OBASE amounted to between two and five percent of total running time. That includes the guides, the tracking, and the migration. The overhead stayed flat whether the system had two workers or thirty-two.

For a representative workload, a key-value store holding ten million records, the baseline system used 12.4 gigabytes of memory. After OBASE reorganized the layout, the same workload ran in 3.5 to 4 gigabytes, a re-

duction of about seventy percent. The application did the same work on the same data. Only the arrangement changed.

8.5 The Messy Middle

Research, as it appears in papers and dissertations, looks like a clean march from question to answer. It is not. The path from "memory is organized badly" to WiscSort and OBASE was full of dead hardware, impossible measurements, and a problem that an entire field had avoided for decades. I want to describe some of that messiness, because it shaped the work in ways the technical sections cannot convey.

The hardware vanished. WiscSort was designed for Intel Optane, a pioneering persistent memory technology that combined the persistence of an SSD (a solid-state drive, the kind of storage that holds your files even when the computer is off) with something approaching the speed of fast memory. I spent years studying its properties, building our five-property model around its quirks, and tuning WiscSort to squeeze every bit of performance from it. Then Intel discontinued Optane. The hardware our entire sorting system was designed for was no longer being manufactured.

This could have been a disaster. Instead, it forced us to ask a more interesting question: which of WiscSort's design principles are specific to Optane, and which generalize to future hardware? To find out, I built emulated devices with different combinations of properties. I could dial up or down the asymmetry between reads and writes, change the interference behavior, adjust the random-read penalty. The result was a map of when each design choice matters. Separating labels from products helps whenever products are larger than labels, regardless of the device. Interference-aware scheduling matters enormously on devices where reads and writes compete for the same internal resources, but adds little when they do not. The worker-pool

controller adapts automatically to whatever concurrency constraints the device imposes.

When CXL memory, a next-generation technology for connecting pools of slower memory to a computer, began to emerge, I found that our model described its properties just as well as it had described Optane's. WiscSort's principles transferred. Losing the hardware I had built for turned out to be the push I needed to build something less fragile.

Measuring the invisible. Before I could fix the mixing problem, I had to prove it existed, and that turned out to be far harder than we expected. Measuring which bytes of memory are actually accessed requires tracking every single load and store instruction a processor executes. For a production workload running for minutes or hours, this generates terabytes of raw trace data. Existing tools could not keep up: they either slowed the workload to a crawl, making the trace unrepresentative, or sampled too sparsely to give accurate per-page utilization numbers.

I had to build my own analysis infrastructure. My trace processing tools were themselves concurrent systems. They parsed billions of memory accesses, annotated each one with its page address and size, and computed utilization statistics across millions of pages, all while keeping their own memory usage and runtime manageable. The engineering required to *study* the problem was nearly as involved as the engineering required to solve it.

When the numbers finally came in, they were more extreme than I had imagined. The Google production traces I analyzed showed that the median page, the typical page in a running workload, used only eight to fifty percent of its capacity. With huge pages, the numbers were even more stark: eighty-five to ninety percent of pages utilized less than ten percent of their space. I had expected some waste. I had not expected that for every dollar spent on fast memory, as little as four cents was going toward data anyone was actually using. These numbers became the foundation of the argument

for OBASE, but reaching them required months of tooling work that never appeared in the final paper.

I analyzed Google's traces because they were among the few production memory traces available to the research community. I expect similar fragmentation in most large-scale deployments (Meta, Amazon, Microsoft, etc), since the root cause, allocators grouping objects by allocation time rather than access pattern, is universal. But confirming this across the industry would require more organizations to share their traces, which remains rare.

The problem nobody had solved. The core technical challenge of OBASE, relocating objects in C and C++ while programs are actively using them, is not new as a concept. Garbage collectors in managed languages like Java have moved objects for decades. They can do this because the language runtime controls every reference to every object. When the garbage collector moves an object, it updates all the references automatically, and the program never sees raw memory addresses to begin with.

C and C++ offer no such infrastructure. A pointer in C is a raw memory address, nothing more. The language provides no way to find every pointer to a given object, no way to intercept when a pointer is followed, and no runtime that could update references on your behalf. These languages have been in widespread use for over fifty years. In all that time, while significant work had gone into automatic *freeing* of memory (detecting when objects are no longer needed), no practical system had been proposed for automatic *relocation* of objects based on how frequently they are accessed.

This was the problem we stared at for a long time. The guide abstraction, the mechanism that finally made relocation possible, emerged from the realization that we did not need to find and update every pointer to an object. We only needed to ensure that every path to an object went through a single stable intermediary. If the intermediary always knew the current address, and if updating the intermediary was atomic (all-or-nothing, with

no half-finished state), then we could move the object and update one value in one place. The compiler could enforce that all access went through guides, and the "try and check" approach could guarantee that no program ever observed a stale address. It sounds straightforward in retrospect. Arriving at it was not.

Making it real. Even after the core mechanism worked, demonstrating that OBASE was practical required building far more than a prototype. Real systems use a remarkable variety of data structures, and each one has its own strategy for handling the fact that multiple workers may try to read or change the same data at the same time. Some systems solve this by closing the entire warehouse while one worker makes a change. Others lock only the specific shelf being touched. Still others use clever tricks to let everyone work simultaneously without locks at all. If OBASE's reorganization conflicted with any of these strategies, it would be useless in practice. I implemented and tested ten different concurrent data structures, from lookup tables used in systems like Redis and NGINX to tree-structured indexes used in databases like PostgreSQL and DuckDB. OBASE had to work with all of them without disrupting their existing coordination strategies.

Proving that the benefits held under realistic conditions meant testing OBASE against recordings of real activity from Meta and Twitter's production caching systems. These recordings captured the full messiness of real use: popularity shifting from one item to another, data being read, updated, and deleted in unpredictable combinations, and item popularity evolving over hours. It is easy to show that a system works on a tidy, artificial test. The real question is whether it works when the patterns are as chaotic as they are in situations closer to practice.

Finally, when OBASE successfully reorganized memory and produced clean cold regions ready for reclamation, I discovered a bottleneck I had not anticipated: the Linux kernel, the core piece of software that sits be-

tween every program and the hardware (PC, phones, etc), acting as the ultimate warehouse manager. When the kernel reclaims a large region of memory by moving it to slower storage, it processes each page one at a time. For every single page, it sends a separate notification to every processor core, telling each one to discard its cached location for that page. For a ten-gigabyte cold region, this meant millions of notifications. The reclamation path, which should have been the easy part, became the slowest step in the pipeline. I modified the kernel to batch these operations, bundling hundreds of pages into a single notification. The fix reduced the number of these inter-processor notifications by over ninety-nine percent and unblocked the entire system. It was a reminder that improving one layer of a complex system often reveals bottlenecks in the next.

8.6 What It All Means

Memory is the single most expensive component in a modern data center, accounting for close to half the cost of a server and a substantial share of its energy consumption. Fast memory carries about twelve times the carbon footprint per bit compared to flash storage, both in the energy needed to manufacture it and the electricity needed to keep it powered around the clock. When companies overprovision fast memory, paying for gigabytes that hold data nobody is accessing, the cost is not just financial. Every wasted gigabyte draws power continuously, adds heat that must be removed by cooling systems that draw still more power, and will eventually be replaced by new hardware that required energy and raw materials to produce. As data centers draw more power worldwide, wasting memory wastes energy too.

The work in this dissertation offers a different path. WiscSort and OBASE are software systems. They require no new hardware, no changes to the operating system's core memory manager, and in the case of OBASE, minimal changes to application code. They work by being smarter about *arrangement*:

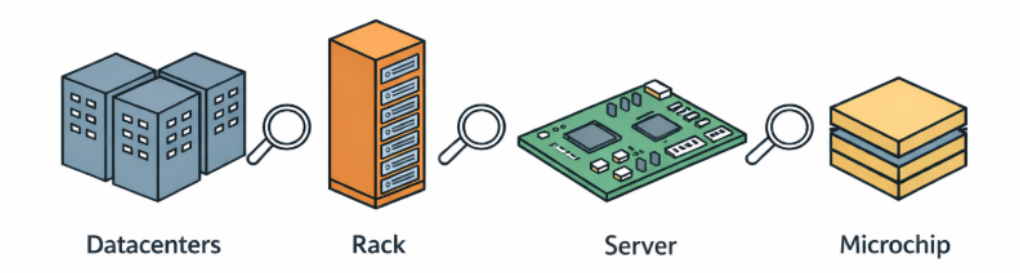

Figure 8.5: **Data organization at every scale.** From datacenters down to individual microchips, the speed of light is constant: accessing nearby data is always faster than reaching for something far away. At every level of this hierarchy, the same principle applies. If you can keep the data that is popular right now physically close to where it is needed, performance improves and resource waste decreases. WiscSort and OBASE demonstrate this principle at the memory and storage layers, but the idea applies wherever data lives.

ensuring that the data the system actually uses is stored where it can be reached quickly, and the data it does not use is stored where it is cheap to keep. WiscSort does this at design time, by understanding the hardware well enough to separate hot data from cold before the work begins. OBASE does it at run time, by watching what the application touches and continuously reshaping the layout in response.

Underneath both systems lies a single insight that I think extends well beyond memory management. The address space, the map that tells the processor where to find each piece of data, is not just bookkeeping. It is a communication channel between the application and the operating system. Today, that channel is mostly silent: applications allocate memory and the operating system manages pages, but neither side tells the other anything about what the data means or how it is used. OBASE shows that when applications speak up, when they organize their data so that pages reflect actual usage patterns, the operating system's existing tools become dramatically more effective without any modification.

This principle could reshape how we think about memory well beyond

the specific problems in this dissertation. Any system that manages objects in memory, whether it is a database, a web browser, or a programming language's garbage collector, faces the same mismatch between what it knows about its data and what the operating system can see. Bridging that gap, letting the software that *understands* the data communicate with the hardware that *moves* it, is a lever that has barely been pulled.

There is more to do. Better hardware support for tracking which data is actually being used could shrink even the small overhead our software-based approach introduces. And the measurement challenge persists: we need more organizations to share real traces of how their systems use memory, so researchers can study the problem as it actually exists rather than as we assume it does.

But even today, the numbers tell a clear story. For every dollar the industry spends on fast memory, the vast majority is wasted on data nobody is touching. WiscSort [1] and OBASE [2, 3] show that we can recover that waste using nothing but smarter arrangement, no new hardware, just a better understanding of what goes where. In an era when data centers consume a growing share of global electricity, using the memory we already have more wisely is not just a scientific challenge. It is a responsibility.

References

[1] Vinay Banakar, Kan Wu, Yuvraj Patel, Kimberly Keeton, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. WiscSort: External Sorting for Byte-Addressable Storage. *Proc. VLDB Endow.*, 16(9):2103–2116, May 2023. ISSN 2150-8097. doi: 10.14778/3598581.3598585. URL `https://doi.org/10.14778/3598581.3598585`.

[2] Vinay Banakar, Suli Yang, Kan Wu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Kimberly Keeton. Tidying Up the Address Space. In *Proceedings of the 3rd Workshop on Disruptive Memory Systems*, SOSP '25, pages 63–72. ACM, October 2025. doi: 10.1145/3764862.3768179. URL `https://doi.org/10.1145/3764862.3768179`.

[3] Vinay Banakar, Suli Yang, Kan Wu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Kimberly Keeton. OBASE: Object-Based Address-Space Engineering to Improve Memory Tiering, 2026. doi: 10.48550/arXiv.2603.00378. URL `https://doi.org/10.48550/arXiv.2603.00378`.