**Chris Down**

# Debunking zswap and zram myths

tl;dr:

- zswap provides graceful degradation under memory pressure. zram provides a hard limit, and is best managed alongside a userspace OOM manager like systemd-oomd if possible.
- If you have fast storage and want graceful degradation, you probably want zswap.
- If you prefer hard limits or have very slow storage, you may prefer zram + a userspace oom killer.
- If you have multiple swap devices, zram's block device architecture creates LRU inversions that can cause serious problems. There is now writeback support, but it can be very difficult to reason about.
- zswap integrates with memory management but can have performance cliffs when the pool fills.
- **In general, with tiered setups (that is, compressed RAM, then falling back to disk for swap) prefer zswap unless you have a strong reason to use zram. For setups with no disk, zram is fine.**

---

I recently received a question from a reader about compressed swap technologies on Linux:

> *I read your articles about memory management (swap) on Linux, finally some words from the expert :) instead of internet experts "you have 32GB - disable it".*
>
> *It'd be nice to have a follow-up about zswap or zram ... :) if they're good (on desktop with 32GB RAM or more), which one is better and why ... I understand how they work (from descriptions), but it's difficult to measure it and compare in real life. Again, a lot of "internet experts" just say "zram - because you don't wear your SSD" ...*

First of all, since I'm writing this article, clearly flattery will get you everywhere ;-)

It is true that there is a lot of confusion and misinformation on the internet about when to use zswap versus zram, and the tradeoffs between them. I've worked on kernel memory management and swap code for the better part of a decade now, and I've seen these technologies evolve and some of the common misconceptions that arise around them.

The truth is that there is not a one size fits all solution. zswap and zram are architecturally quite distinct, and deciding which to use when requires some nuance and thought.

## Architectural differences

The fundamental difference between zswap and zram lies in where they sit in the kernel's storage hierarchy.

- zram acts as a compressed block device, essentially a virtual disk in RAM. When a process needs to swap, the kernel treats swap on zram like it does on any other block device, sending I/O requests through the block layer. Once zram fills up, it's just another storage device that's reached capacity. There's no automatic mechanism to move data elsewhere.
- zswap, on the other hand, is more integrated with the memory management subsystem overall. It acts as a compression layer that sits in front of your disk swap. When a process needs to swap, zswap intercepts the page before it reaches disk, decides whether to compress it, and if it does, it stores it in a memory pool. When that pool fills up, zswap uses its own heuristics to evict the least recently used pages to your backing swap device, aiming to keep your hot data in the compressed RAM cache.

The result is basically that zram provides a hard capacity limit, whereas zswap provides a kind of automatic tiering between faster (zswap) and slower swap (disk swap) to gracefully degrade as memory pressure increases.

So what are the ramifications? Well, for most workloads, the choice between zswap and zram depends on what you value more: graceful degradation or hard limits. But to understand why, let's look at how these technologies actually work at the kernel level, and why their architectural differences create fundamentally different performance characteristics.

## *zram's block device architecture*

zram creates a compressed block device that acts as standalone swap. The typical procedure for setting up swap on zram is to:

1. Create a zram device by loading the module with `modprobe zram`
2. Set `comp_algorithm` and `disksize` to taste
3. `mkswap` on `/dev/zram0`
4. `swapon` on `/dev/zram0`

You may notice that this looks pretty much exactly like what you would do if you were setting up swap on a partition, and that's not a coincidence: the kernel sees it as just another storage device through the block layer:

```
/*
 * Handler function for all zram I/O requests.
 */
static void zram_submit_bio(struct bio *bio)
{
    struct zram *zram = bio->bi_bdev->bd_disk->private_data;

    switch (bio_op(bio)) {
```

```
    case REQ_OP_READ:
        zram_bio_read(zram, bio);
        break;
    case REQ_OP_WRITE:
        zram_bio_write(zram, bio);
        break;
    case REQ_OP_DISCARD:
    case REQ_OP_WRITE_ZEROES:
        zram_bio_discard(zram, bio);
        break;
    default:
        WARN_ON_ONCE(1);
        bio_endio(bio);
    }
}
```

*zram_submit_bio() from Linux 6.18*

This block device approach makes zram perfect for the average embedded system: it's completely self-contained, and it doesn't depend on any disk storage. When you're on a Raspberry Pi with an SD card or in a diskless container, zram gives you some amount of memory offload without any external dependencies. All pretty reasonable so far.

But when you *do* have disk storage available (like an SSD), zram's block device architecture creates both opportunities and constraints. The fact that the kernel is basically unaware of the nature of the backing device that provides the swap kernel (that is, it treats zram as just another disk) has fairly significant implications we'll explore in a moment.

## *zswap's memory management integration*

By comparison, zswap doesn't create a block device, and instead integrates directly into the kernel's memory management subsystem. When the kernel needs to swap out a page, it calls `swap_writepage()`, which gives zswap first dibs to intercept it:

```
int swap_writeout(struct folio *folio, struct swap_iocb **swap_plug)
{
    /* ... */

    if (zswap_store(folio)) { /* zswap has called bagsy on the page
        count_mthp_stat(folio_order(folio), MTHP_STAT_ZSWPOUT);
        goto out_unlock;
    }
    if (!mem_cgroup_zswap_writeback_enabled(folio_memcg(folio))) {
        folio_mark_dirty(folio);
        return AOP_WRITEPAGE_ACTIVATE;
    }
    __swap_writepage(folio, swap_plug);
    return 0;
out_unlock:
    folio_unlock(folio);
    return ret;
}
```

*swap_writeout() from Linux 6.18*

If `zswap_store()` returns `true`, the page has been stored in compressed RAM and never touches disk. Only if zswap declines (or isn't enabled) does the kernel fall back to writing to the backing swap device.

Here's what `zswap_store()` does internally:

```
bool zswap_store(struct folio *folio)
{
    /* ... */

    /* Check if we've hit pool size limits */
    if (zswap_check_limits())
        goto put_objcg;

    /* Get the current compression pool */
    pool = zswap_pool_current_get();
    if (!pool)
        goto put_objcg;

    /* Try to compress and store each page in the folio */
    for (index = 0; index < nr_pages; ++index) {
        if (!zswap_store_page(page, objcg, pool))
            goto put_pool;
    }

    ret = true;  /* Success! */

put_pool:
    zswap_pool_put(pool);
put_objcg:
    obj_cgroup_put(objcg);

    /* If we failed because pool was full, queue work to shrink it 
    if (!ret && zswap_pool_reached_full)
        queue_work(shrink_wq, &zswap_shrink_work);
check_old:
    return ret;
}
```

*zswap_store() from mm/zswap.c*

The shrink worker automatically evicts cold pages to disk when zswap fills up:

```
static void shrink_worker(struct work_struct *w)
{
    struct mem_cgroup *memcg;
    int ret, failures = 0, attempts = 0;
    unsigned long thr;

    /* Reclaim down to the accept threshold */
    thr = zswap_accept_thr_pages();

    do {
        /* ... memcg iteration logic ... */

        ret = zswap_shrink_memcg(memcg, thr);
        if (ret == -ENOENT) {
            /* No writeback candidate found, try next memcg */
            failures++;
        } else if (ret == -EAGAIN) {
            /* Writeback is disabled or in progress */
            failures++;
        } else if (ret) {
            /* Other errors */
            failures++;
        } else {
            /* Success */
            attempts++;
            failures = 0;
        }

        /* ... continue to next memcg ... */
    } while (attempts < MAX_RECLAIM_RETRIES &&
             failures < MAX_RECLAIM_RETRIES);
}
```

*shrink_worker() from Linux 6.18*

This tight integration with the rest of the memory management subsystem, and the fact that other mm code is aware of the nature of this storage is what makes zswap work differently from zram. zswap acts as a transparent compression layer *in front of* your SSD swap, not as a separate storage tier, like how zram does it. When the pool fills up, it automatically triggers the shrinker to evict cold pages to disk. But as we'll see, this also has its own nuances and failure modes under extreme pressure.

## LRU inversion

Just as one example, here is one very important architectural issue that can affects zram when tiering alongside regular disk swap. In it, the story goes something like this:

1. A page is ready to be swapped, and goes into [give kernel function]
2. There's no zswap, only zram, so the kernel just looks down the list of swap devices
3. zram is configured with the highest swap priority, so it gets chosen as long as it has space.

This all might seem fine on paper. But the problem is that when the kernel allocates swap space across multiple devices, it uses a priority-based allocation system:

```
/* Rotate the device and switch to a new cluster */
static void swap_alloc_slow(swp_entry_t *entry, int order)
{
    unsigned long offset;
    struct swap_info_struct *si, *next;

    spin_lock(&swap_avail_lock);
start_over:
    plist_for_each_entry_safe(si, next, &swap_avail_head, avail_list
        /* Rotate the device and switch to a new cluster */
        plist_requeue(&si->avail_list, &swap_avail_head);
        spin_unlock(&swap_avail_lock);
        if (get_swap_device_info(si)) {
            offset = cluster_alloc_swap_entry(si, order, SWAP_HAS_CA
            put_swap_device(si);
            if (offset) {
                *entry = swp_entry(si->type, offset);
                return;
            }
            if (order)
                return;
        }

        spin_lock(&swap_avail_lock);
        /* ... continue to next device if this one is full ... */
    }
}
```

*swap_alloc_slow() from Linux 6.18*

Higher priority devices get used first. That sounds fine, right? Of course it does, that's why users configure it this way all the time. But such users have unwittingly created a trap that grows more and more likely with more uptime.

So why is that? Well, since the swap on the zram device has the highest priority, the kernel prefers zram for all allocations. When zram fills up, it switches to the disk based swap for all future allocations.

That means that without intervention, your precious zram gets filled with whatever pages *happened to be swapped out first*. That is usually totally inversely correlated with the pages that you actually need *now*.

In a typical desktop session, these pages are usually cold, initialisation time data that is evicted early to make room for (say) the browser you just opened. These cold pages then permanently occupy the fast zram. Meanwhile, as your session continues and memory pressure persists, the newer, potentially "hotter" pages (like recent browser tabs you are actively switching between) are forced to spill over to the lower-priority device: the slow mechanical disk or SSD.

This is LRU inversion. Your fastest storage tier is effectively clogged with the coldest data with no way to evict it, and this forces your active working set onto the slowest storage tier, completely negating the performance benefits zram was meant to provide.

## *zram writeback and its limitations*

Now, that's where the discussion would end if I was writing this a few years ago. :-) But since kernel 4.14, zram has has writeback support, which is an attempt to address this problem.

With writeback configured, zram can write back pages that are idle or do not compress well. This means modern zram setups *can* implement tiering, but:

1. It requires manual configuration rather than happening automatically in the kernel reclaim path, and
2. It requires you to think about and set up a reasonable method to go about the writeback and heuristics.

Maybe that doesn't sound so difficult. Allow me to try to convince you otherwise.

To achieve similar behavior to zswap, where incompressible or idle pages are moved to disk, you must create your own solution. One problem is that you cannot simply point zram writeback at a swapfile or your existing swap partition. The writeback interface requires a dedicated, unformatted block device. With `zram-generator`, that looks something like this:

```
[zram0]
zram-size = ram / 2
writeback-device = /dev/sda4
```

Another problem is that you must repartition your disk to create a dedicated partition for zram backing, or manage loopback devices (which adds overhead). You also cannot share this space with the system's hibernation swap or other data easily.

An even bigger hurdle is actually going about the flushes. Even with the device attached, zram does not write out pages to it automatically, and the kernel has no internal heuristic to decide when to move data from zram to the backing device, because zram is not integrated enough into the mm subsystem to achieve that effectively.

Instead, you must create a systemd timer or a cron job to manually trigger this flush, and even this is not that easy.

For example, to flush out pages that zram could not compress, it is as simple as:

```
echo huge > /sys/block/zram0/writeback
```

...but to flush idle or cold data, it is even more complex. Because zram is agnostic of the fact that it has swap on it, as opposed to any other kind of data, zram doesn't inherently track LRU age in a way that allows simple eviction. You first have to tell the kernel to mark pages as idle, and then tell zram to write them out.

```
echo 3600 > /sys/block/zram0/idle # 1h
echo idle > /sys/block/zram0/writeback
```

As well as the complexity, zram is architecturally at quite a disadvantage compared to zswap's native LRU tiering.

For example, in zram, this age check is a one-time event. When you run your script or timer, it takes a snapshot of the current state. If a page becomes cold 5 minutes later, it stays in RAM until you run the script again. There's no connection to the reclaim process or shrinkers, and if memory pressure suddenly rises, it may be too late to run your script.

Compare that with zswap, where the LRU list is evaluated as part of the normal memory lifecycle. As soon as memory pressure rises, the kernel looks at the live list of pages as part of reclaim and evicts the oldest ones based on that behaviour. It is, in general, more resilient and integrated with the normal memory lifecycle.

There is also a granularity problem. With zram, you have to guess a magic number to perform eviction based on time (like 24 hours). If you guess too high, you waste RAM. If you guess too low, you flush data that you might have actually wanted. The system does what you say, and without extensive profiling over time, it is hard to know what to tell it to be effective.

By comparison, in zswap, there is no magic number, and it dynamically balances the LRU based on pressure. If you have plenty of RAM and there's no pressure, it keeps data indefinitely. If you are starving for RAM, it aggressively evicts the oldest data, whether it's 24 hours old or 24 minutes old.

So, I would argue that while the LRU inversion problem in zram has *some* kind of solution, it's not exactly an intuitive or easily usable one, and for most users, the way zswap handles LRU inversion avoidance is significantly better.

## zswap's automatic tiering (and its performance cliffs)

As we discussed, zswap is much more tightly integrated with the memory management subsystem than zram is, and one of the main things it benefits from that is the ability to do proper tiering through the shrinker interface. Here's the code, with some comments added to aid those unfamiliar with this code:

```
static unsigned long zswap_shrinker_count(struct shrinker *shrinker,
        struct shrink_control *sc)
{
    /* zswap shrinker_count basically answers the question of how ma
     * should evict from zswap to the backing swap device. */

    /* This is how often we had to fetch data from slow disk recentl
     * this to avoid thrashing. */
    atomic_long_t *nr_disk_swapins =
        &lruvec->zswap_lruvec_state.nr_disk_swapins;

    /* ... */

    /* Subtract from the lru size the number of pages that are recen
     * in from disk. The idea is that had we protect the zswap's LRU
     * amount of pages, these disk swapins would not have happened.
    nr_disk_swapins_cur = atomic_long_read(nr_disk_swapins);
    do {
        if (nr_freeable >= nr_disk_swapins_cur)
            nr_remain = 0;
        else
            nr_remain = nr_disk_swapins_cur - nr_freeable;
    } while (!atomic_long_try_cmpxchg(
        nr_disk_swapins, &nr_disk_swapins_cur, nr_remain));

    nr_freeable -= nr_disk_swapins_cur - nr_remain;
    if (!nr_freeable)
        return 0;

    /* Scale eviction by compression ratio. If compression is good (
     * we evict fewer pages to avoid wasting I/O for small gains. */
    return mult_frac(nr_freeable, nr_backing, nr_stored);
}
```

*zswap_shrinker_count() from Linux 6.18*

This means that with zswap, the kernel itself is helping keep your compressed RAM working towards containing the hottest data, and cold pages automatically trickle down to the SSD. The shrinker also accounts for compression ratios – that is, the better zswap compresses, the fewer pages it evicts to avoid unnecessary I/O.

But there is, as always, a catch. When zswap hits its `max_pool_percent` limit, it faces a choice:

1. It can accept the page. To do this, it needs to evict something to disk to make room, which adds latency.
2. It can also reject the page and send it straight to uncompressed disk, which bypasses the cache.

This means that if the backing disk is slow or the system is under heavy memory pressure, zswap can start rejecting pages and bypassing the cache entirely. This creates a performance cliff where the system suddenly drops swap performance from the magnitude you might expect from RAM access to the magnitude you might expect from disk access without much warning. This is not worse than zram with tiering, but it is something to bear in mind.

Also, some advice on allocator selection. When configuring zswap, the choice of allocator matters significantly for this cliff. Prefer zsmalloc over the older z3fold or zbud allocators. zsmalloc achieves much higher compression ratios by grouping similar objects, whereas the older allocators use fixed-size objects that tend to waste space. It also interacts better with writeback throttling.

You can enable it with:

```
echo zsmalloc > /sys/module/zswap/parameters/zpool
```

As of writing this post in early 2026, z3fold and zbud are deprecated, but may still be the default on some distro kernels.

## Performance characteristics and trade-offs

Before we discuss when to use each technology, let's understand their performance implications. Both technologies trade CPU cycles for reduced I/O, but they have different overhead profiles and failure modes.

### *Handling incompressible data*

There is a subtle but important difference in how the two technologies handle data that doesn't compress well:

zswap can detect incompressible pages during compression and reject them, sending them straight to disk. This saves both RAM (by not storing poorly compressed data) and CPU cycles (by not repeatedly trying to compress incompressible data). You can see how often zswap has done that in the `reject_compress_poor` counter in `/sys/kernel/debug/zswap/`.

By comparison, by default zram compresses everything regardless of compression ratio. zram tracks these poorly-compressed pages in its `huge_pages` statistic, but without specific writeback configuration, zram will store even 4KB pages that compress to 3.9KB, wasting both memory and CPU.

This means zswap has better worst-case behavior for workloads with lots of incompressible data (encrypted files, compressed media, already-compressed application data), though the difference is often negligible in practice for typical mixed workloads.

### *SSD wear considerations*

Another reason some people say to prefer zram over zswap is that they believe that it reduces SSD wear – that is to say, they believe using it reduces disk I/O.

But this is a mistake. RAM is finite. If you are filling your RAM with some kind of data, eventually, when all of your RAM is used, the data needs to go somewhere.

You might think you will never use all of your RAM. But on Linux, we don't actually leave RAM empty. The kernel follows the philosophy that unused RAM is wasted RAM, and automatically filling any slack space with the page cache and other nice to have things, that is, things like copies of files, libraries, and disk data to speed up future access.

As part of this, the kernel daemon `kswapd` proactively wakes up to reclaim memory when free space dips below certain watermarks and works to balance memory usage. In the ideal case, we want to always have pages available to immediately allocate without having to sleep for reclaim, so it manages pressure as a normal state of operation to ensure there is always a buffer for immediate allocation.

**zram-only removes disk swap I/O, but it can shift pressure onto the page cache. Under memory pressure, more file cache may be dropped (leading to rereads) or written back (if dirty). With disk-backed swap (or zswap), the system can often evict cold anonymous pages instead, which may reduce cache churn, and thus reduce I/O. That means that zram can actually *increase* total disk I/O if not well managed.**

So how can that be? Well, memory in most cases on both servers and desktop is dominated by two types of pages. One is anonymous pages, like your program heap and stack data. The other is file pages, that is, the disk cache. If you use zram without a physical backing device, you effectively lock all anonymous data in RAM. When memory pressure hits, the kernel has no choice but to aggressively evict the file cache to make room.

If those evicted file pages are "dirty" (that is, they contain modified data), the kernel is forced to write them to the SSD to free up space and make forward progress. Even if they are "clean" (that is, they are unmodified), they are dropped, forcing the SSD to read them again the next time they are needed. By refusing to swap out cold, unused anonymous data to a physical disk via zswap or swap partition, you strangle the page cache. This forces the system to constantly flush and re-read active files.

The real goal here is to load the right things into RAM, keeping the active "working set" in memory, not to overuse zram.

And, in defense of zswap, it too also dramatically reduces SSD wear by acting as a write-reduction filter. It absorbs high-frequency page-out/page-in transients in RAM, compressing data before it ever touches the disk. Only truly cold data that survives the cache gets written there.

Modern SSDs are also capable of typically handling hundreds of terabytes of writes. Consumer SSDs typically offer 150-600 TB TBW. In 2025, this whole conversation is largely moot anyway unless you are using very cheap eMMC storage, and even then zram may not be your best bet.

## *Performance under memory pressure*

The only place most users will see real performance differences between zswap and zram is in their failure modes, which are extremely different, and will suit different audiences.

Both zswap and zram compress pages in RAM under normal operation, and there's very little difference in the overheads that can be seen outside of benchmarking.

However, under load, when using zswap:

1. When the pool is filling, there is automatic LRU based eviction to disk
2. When the pool is full, we can can start rejecting pages and bypassing the cache, so there is a performance cliff
3. Under extreme pressure the system can degrade to disk swap speed

With zram, by comparison:

1. When the devie is filling, it simply continues accepting pages until full
2. When the device is full, it switches to next priority swap device (or OOMs if there is none). There is no automatic eviction.
3. Under extreme pressure, when there's no other backing device, the system tends to OOM rather than thrashing.

You might think hey, if the system is swapping heavily to disk, desktop responsiveness is already ruined. I'd rather have the system OOM kill a process than slowly thrash the user to death. But there is a dangerous nuance here that is often overlooked – the kernel OOM killer is not even close to instantaneous.

As I went over in my SREcon talk, relying on the kernel's built-in OOM killer to save responsiveness is often a losing battle. The kernel doesn't actually know when it is out of memory, it only knows that it has tried very hard to reclaim memory and failed.

Before the OOM killer is ever invoked, the kernel enters a cycle of aggressive reclaim:

- It scans LRU lists looking for clean pages to drop.
- It attempts to flush dirty pages to disk (if writeback is available).
- It cycles through various memory types trying to free anything.

This process can take seconds or even minutes. During this time, your application is suspended, and the system appears to hang. By the time the OOM killer actually fires, the user has likely already experienced significant unresponsiveness, and the system may be locked up to the point that the user can do very little about it.

The kernel OOM killer is also very imprecise. It uses user-provided scores to decide who to kill, which often results in simply killing the largest process, rather than necessarily the process that is actually leaking memory.

## zram on Fedora

So if I've spent so long going over why zram is so difficult, why are distributions like Fedora defaulting to zram-only setups even on desktops with fast SSDs?

The reason is that they try to avoid using the kernel OOM killer at all. Modern Fedora uses systemd-oomd in order to get ahead of shortages and apply a userspace policy to decide what to do about it. They want a hard limit, and they want the system to kill something to make forward process rather than degrade performance.

In order to do this one *has* to have userspace tools (like systemd-oomd or Android's lmkd) that monitor Pressure Stall Information (PSI). These tools can see when memory pressure is slowing the system and kill processes proactively based on highly nuanced policy before the kernel enters its pathological reclaim loop. Without those tools, you will likely still experience the system hangs you were trying to avoid.

This is a fundamentally different philosophy from zswap's graceful degradation approach. Fedora doesn't want to smoothly degrade performance as memory pressure increases using all available storage tiers. Instead, they want to maintain performance at all costs and hard fail if one exceeds the available capacity.

Fedora also sidesteps the LRU inversion issues I mentioned earlier by having only one swap device. There is no swap partition or swap file on the disk, so because there is no second order storage, there is no relationship to invert.

This is also a totally valid way of going about things. It is totally defensible to say you'd rather your browser tabs load slowly from disk than have the system kill my work, and it's also totally defensible to conversely argue that you'd rather have the system kill a process than hang for 30 seconds. But you have to decide.

For interactive desktop workloads, the Fedora approach has merit. A desktop user experiencing significant swapping activity is probably already experiencing low responsiveness, and may be better served by a userspace OOM daemon terminating the problematic process. Compared to a server, it may be much less destructive to restart the killed process rather than waiting for swap I/O.

## If I use zram, how should I size the device?

In the naive case, with zram, you have to guess the device size upfront. If you guess too small, you waste potential. If you go too large you risk OOM killing, or cause unnecessary minor faults.

So how does Fedora size it? Well, they size it up to 100% of your RAM. Job's a goodun.

Okay, maybe that does require a little more explanation. :-)

Fedora takes an aggressive approach here. They size the zram device to 100% of your physical RAM, capped at 8GB. You may be wondering how that makes any sense at all – how can one have a swap device the size of one's RAM? And surely to read a page from zram, I have to decompress it into main RAM, right?. If zram is full, where do I put the decompressed page?

Fedora solves this with a bit of educated gambling. First, zram is thin provisioned. Until pages are actually faulted, there's no memory use. So a zram device sized to 100% with nothing in it occupies no space except for the space used for bookkeeping.

In addition to that, they are betting that your data compresses well, let's say ratio 3:1. So then, a 100% sized zram device will only physically occupy one third of RAM. This leaves 66% of RAM free for the OS and decompression buffers.

Then then use systemd-oomd to watch memory pressure. If it sees zram physically filling up RAM, it kills something based on policy before you hit the deadlock wall where where there wouldn't be enough space to compress.

## How to decide which to use

So that's a lot of information and nuance. Based on that, how should one decide what to use?

I would argue that one should probably should choose zswap when tiering with disk based swap. It is much more tightly integrated with the rest of the memory management subsystem, has much better heuristics around reclaim and eviction, and generally has better behaviour at the extreme edge cases. It also handles hibernation transparently if you care about that. Disk based swap still has significant upsides in many cases, even if you have a lot of RAM, so I would suggest using it if possible.

But, as Fedora shows, zram is also quite usable on the desktop, as long as you have the commensurate tooling to support it, like `systemd-oomd`, and avoid having any other swap devices. By going this way you may not be getting the theoretical maximum out of your memory, but you also are avoiding some of the gnarly edge cases from tiering overall. So it's swings and roundabouts.

On embedded systems, zram makes significantly more sense. It is extremely simple, extremely predictable, and in those environments usually manually sizing things makes more sense, and the workloads are much better understood and regulated.

On servers, in my opinion zram is a hard sell for a general audience. You are effectively telling the kernel to crash a service rather than let it run slowly for a few minutes, which might be okay for some services, but for some services restarting can be very expensive or disruptive. It can also manifest even more poorly at scale, where a minor memory spike across a fleet can turn into a mass outage rather than being absorbed as a temporary latency increase. It also doesn't work well with tiering. I can imagine use cases for it on the server too, but they are not well generalised or easy to recommend.

Both technologies are valid, and both have their place. The real answer depends on your performance philosophy, your workload, and your tolerance for complexity. Hopefully this post has helped you to understand the tradeoffs better so you can make an informed decision.

---

Chris Down
PGP: 0xDF8D21B616118070

Questions or comments?
My e-mail is chris@chrisdown.name.