Understanding RAID: How performance scales from one disk to eight

Ever wondered how performance scales with number of disks? Read on, friend.

JIM SALTER · APR 16, 2020 1:00 AM · 318

→ Behold—16TB of storage stacked on a workbench in an unwieldy, eight-high spiral. Don't try this at home, kids; photography and system administration don't mix very well. Credit: Jim Salter

TEXT SETTINGS

One of the first big challenges neophyte sysadmins and data hoarding enthusiasts face is how to store more than a single disk worth of data. The short—and traditional—answer here is RAID (a Redundant Array of Inexpensive Disks), but even then there are many different RAID topologies to choose from.

Most people who implement RAID expect to get extra performance, as well as extra storage, out of all those disks. Those expectations aren't always rooted very firmly in the real world, unfortunately, but since we're all home with time for some experimentation, we hope to shed some light on how to plan for storage performance—not just the total number of gibibytes (GB) you can cram into an array.

A quick note here: Although readers will be interested in the raw numbers, we urge a stronger focus on how they relate to one another. All our charts relate the performance of RAID arrays at sizes from two to eight disks to the performance of a single disk. If you change the model of disk, your raw numbers will change accordingly—but the relation to a single disk's performance will not for the most part.

Equipment as tested

→ (left) Yes, I work in a largely unfinished basement. At least I've got windows out into the yard. Don't ill me. Jim Salter

→ (right) This is the Summer 2019 Storage Hot Rod, with all twelve bays loaded and hot. The first four are my own stuff; the last eight are the devices under test today. (The machine above it is banshee, my Ryzen 7 3700X workstation, in an identical 12-bay chassis.) Jim Salter

We used the eight empty bays in our Summer 2019 Storage Hot Rod for this test. It's got oodles of RAM and more than enough CPU horsepower to chew through these storage tests without breaking a sweat.

SPECS AT A GLANCE: SUMMER 2019 STORAGE HOT ROD, AS TESTED	
OS	Ubuntu 18.04.4 LTS
CPU	AMD Ryzen 7 2700X—$250 on Amazon
RAM	64GB (2 SEC 32GB UDIMM kit)—$460 at Amazon
STORAGE ADAPTER	LSI 9300-8i 8-port Host Bus Adapter—$150 at Amazon
STORAGE	8x 12TB Seagate Ironwolf—$320 ea at Amazon
MOTHERBOARD	Asrock Rack X470D4U—$260 at Amazon
PSU	EVGA 850GQ Semi Modular PSU—$150 at Google Shopping
CHASSIS	Rosewill RSV-L4112—Typically $260, currently unavailable due to CV19

The Storage Hot Rod's also got a dedicated LSI 9300-8i Host Bus Adapter (HBA) which isn't used for anything but the disks under test. The first four bays of the chassis are filled with my own backup data—that's why they were idle during all tests here, and are attached to the motherboard's SATA controller, entirely isolated from our test arrays.

How we tested

As always, we used fio to perform all of our storage tests. We ran them locally on the Hot Rod, and we used three basic random-access test types: read, write, and sync write. Each of the tests was run with both 4K and 1M blocksizes, and I ran the tests both with a single process and iodepth=1, and with eight processes with iodepth=8.

For all tests, we're using Linux kernel RAID, as implemented in the Linux kernel version 4.15, along with the ext4 filesystem. We used the `--assume-clean` parameter when creating our RAID arrays in order to avoid overwriting every block of the array, and we used `-E lazy_itable_init=0,lazy_journal_init=0` when creating the ext4 filesystem to avoid contaminating our tests with ongoing background writes initializing the filesystem in the background.

Kernel RAID vs hardware RAID

We do not have side-by-side tests with a hardware RAID adapter here, so you'll need to take our word for it when we tell you that hardware RAID is not magic. We have privately tested Linux kernel RAID versus popular professional, dedicated eight-port hardware RAID cards several times over the years, however.

For the most part, kernel RAID significantly outperforms hardware RAID. This is due in part to vastly more active development and maintenance in the Linux kernel RAID than is devoted to firmware for the cards. It's also worth noting that a typical modern server has tremendously faster CPU and more RAM available to it than a hardware RAID controller does.

The one exception to this rule is that some hardware RAID controllers have a battery-backed cache. These cards commit sync write requests to the onboard, battery-backed cache instead of to disk, and they lie to the operating system about it. Cached, synchronous writes aggregate then trickle out from the controller's cache to disk. This works—and performs—just like asynchronous writes that are aggregated and committed by the operating system itself.

Asynchronous writes greatly outperform synchronous writes, and so this represents a significant boost to such a controller's performance. The cited relies on the battery-backed cache's ability to preserve survival of the cached data across power outages. This is, for the most part, like putting the entire server on a UPS and using the amusingly yet appropriately named libeatmydata, which causes the operating system to lie about the results of `fsync()` calls.

A word to the wise: If the battery fails in a RAID controller and the controller does not detect it, corruption can and will result after power outages, since the card is still lying to the operating system and applications when they request assurances that data has been committed safely to disk. If the controller does proactively detect the battery failure, it simply disables on-card write aggregation entirely—which returns sync write performance to its true, far lower level.

In our experience, administrators are overwhelmingly likely not to notice when a hardware controller's cache batteries fail. Frequently, those administrators will still be operating their systems at reduced performance and reliability levels for years afterward.

One final warning about hardware RAID controllers: It's difficult to predict whether a hardware RAID array created under one controller will import successfully to a different model of controller later. Even if the model of controller remains the same, the user interfaces of the management applications or BIOS/UEFI routines used to import arrays are frequently written in incredibly unclear language.

We find that with hardware RAID, it's frequently difficult to tell whether you're nuking your array or importing it safely. So in the event of a controller failure and replacement, you may end up sweating bullets, YOLOing, and hoping. Caveat imperator.

Motherboard RAID

Many enthusiast motherboards offer their own RAID "controllers" on board. These should in no circumstances be confused with true hardware RAID.

Motherboard RAID "controllers" are frequently, and disparagingly, referred to as "fakeRAID." Unlike hardware RAID controllers, they do not have their own CPU or RAM, and they rely on the system's own CPU and RAM.

In the Linux world, fakeRAID generally still means kernel RAID, for the most part. The motherboard "controllers" calls are intercepted by the Linux kernel, handled similarly, and simply exposed as `/dev/sd` rather than the standard `/dev/md` used by normal kernel RAID.

This still represents an additional, unnecessary layer of abstraction which does not improve performance, and may hamper maintenance and/or array portability between one motherboard and another.

In the Windows world, motherboard fakeRAID is a far larger problem. Here, the system CPU and RAM are still used to operate the array—but the code doing so is a proprietary driver provided by the motherboard manufacturer. Windows fakeRAID array creation and maintenance must generally be done either within the BIOS or UEFI of the motherboard, or in a proprietary—and usually quite crappy—application provided by the motherboard vendor.

These motherboard fakeRAID arrays under Windows will generally not survive a move from one motherboard to another—lose your motherboard, lose your array. Again, caveat imperator.

Mebibytes (MiB) vs Megabytes (MB)

A Kilobyte is 1,024 bytes, where a Kilobyte is 1,000 bytes. Similarly, a Mebibyte is 1,024 Kilobytes (1,048,576 bytes), where a Megabyte is 1,000 Kilobytes (1,000,000 bytes).

This is, in our opinion, one of the most common sources of confusion in the IT industry in general. For longer than most readers have been alive, operating systems have represented both RAM and storage in powers-of-two units, but referred to them using standard powers-of-ten metric prefixes.

Meanwhile, storage vendors gleefully label their disks in metric prefixes, despite the individual block or page sizes of the disks being in powers-of-two prefixes. This misleading practice allows vendors to inflate the relationship of capacity to price: an 8TB disk is, technically, accurately labeled—even though its capacity as reported by the operating system will only be 7.28TiB.

We've chosen to use the metric prefixes and Mebibytes and Kilobytes, just like your operating systems use. Any time you see a bare "K" or "M" as a unit here, it refers to Kilo- or Mebi-, not Kilo- or Mega-.

IOPS vs throughput

→ (left) In this test, we're looking at 4K reads with a single process. Don't pay too much attention to the numbers yet—notice the shape, and look at the next chart in this gallery. Jim Salter

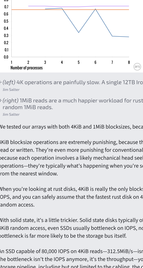
→ (right) Look familiar? That's because 4K single-process read is a test that bottlenecks on IOPS, with no throughput bottleneck. So the relationship is simple and direct: throughput=IOPS*16KiB/1024KiB/MiB. Jim Salter

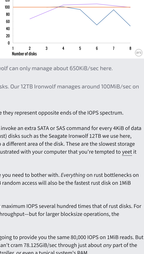
It's tough to decide how to represent performance in a storage article aimed at mass consumption and education. On the one hand, throughput—represented in MiBs/s—is a more immediately familiar measure to most people. On the other hand, we're testing rust disks on a fast system, and literally every test we're doing today bottlenecks on IOPS—I/O Operations Per Second, the number of individual operations that your storage can fulfill.

A rust disk—or an array of them—doing 4KiB reads will always bottleneck on IOPS. A solid state disk or array doing 1MiB reads will usually bottleneck on throughput. It can't get all the data to the CPU any faster than the SATA or SAS bus can carry it. Storage professionals usually talk in units that a particular workload bottlenecks on—so IOPS for punishing small-blocksize operations, and throughput for easy large-blocksize operations.

We decided to go with throughput simply because it will be so familiar to most people, and IOPS won't mean much to anyone who isn't living and breathing storage administration professionally. But if you take a quick look at the two charts above, you'll see that they relate quite directly: throughput=IOPS*blocksize.

4KiB vs 1MiB blocksize





→ (left) 4K operations are painfully slow. A single 12TB Ironwolf can only manage about 650KiB/sec here. Jim Salter

→ (right) 1MiB reads are a much happier workload for rust disks. Our 12TB Ironwolf manages around 100MiB/sec on random 1MiB reads. Jim Salter

We tested our arrays with both 4KiB and 1MiB blocksizes, because they represent opposite ends of the IOPS spectrum.

4KiB blocksize operations are extremely punishing, because they invoke an extra SATA or SAS command for every 4KiB of data read or written. They're even more punishing for conventional (rust) disks such as the Seagate Ironwolf 12TB we use here, because each operation involves a likely mechanical head seek to a different area of the disk. These are the slowest storage operations—they're typically what's happening when you're so frustrated with your computer that you're tempted to vent it from the nearest window.

When you're looking at rust disks, 4KiB is really the only blocksize you need to bother with. Everything on rust bottlenecks on IOPS, and you can safely assume that the fastest rust disk on 4KiB random access will also be the fastest rust disk on 1MiB random access.

With solid state, it's a little trickier. Solid state disks typically offer maximum IOPS several hundred times that of rust disks. For 4KiB random access, even SSDs usually bottleneck on IOPS, not throughput—but for larger blocksize operations, the bottleneck is far more likely to be throughput than IOPS.

An SSD capable of 80,000 IOPS on 4KiB reads—312.5MiB/s—isn't going to provide you the same 80,000 IOPS on 1MiB reads. But the bottleneck isn't the IOPS anymore, it's the throughput—you can't cram 78.125GiB/sec through just about any part of the storage pipeline, including but not limited to the cabling, the controller, or even a typical system's RAM.

Random access vs sequential

If you use a tool such as fio or iozone to literally read every block from a storage device in sequential order, you've got a sequential workload. However, nearly any use of an actual filesystem will transform a theoretically sequential workload into a large block random access workload.

A file has metadata as well as data, and they aren't usually stored entirely in the same place. Filesystems get fragmented, requiring some portions of a file to be stored in one block of free space and other portions in another. Equally important: Very few real-world apps actually do even close to a single thread of operation both want to access storage at the same time, the operating system must rapidly switch back and forth between them in order to minimize the latency each endures.

All tests performed here are random access, because nearly any real-world storage workload is random access. Truly sequential workloads may be slightly larger than the 1MiB random-access tests below, but not by much. Since sequential access is, for the most part, a happy myth admins like to tell one another so they can look at comfortingly large numbers, we don't test anything closer to truly sequential than 1MiB random access.

Read vs write

In private storage testing, we generally don't bother with reads—they're a pain in the butt to test accurately, since any cache hits are several orders of magnitude faster than the actual storage.

You can typically expect most storage media to be slightly faster in reading data than it is at writing it, but while the storage can "finesse" writes to try to minimize fragmentation and unnecessary consumption of IOPS, it must take the reads from where they already lie on the disk.

We test both reads and writes here, with caches carefully flushed and test sizes carefully calculated to minimize cache hits, because we know readers will be curious. Just remember that observed read performance will vary wildly in real life, because a cached read completes 100x faster (or more!) than an uncached read and because reading data that was written fragmented will proceed far slower than reading data that was written contiguously.

Synchronous writes vs asynchronous

↑ 4KiB sync writes are the most painful possible workload—closely followed by 4KiB (uncached) reads. When the operating system is forced to aggregate writes over time and flush them in batches, performance improves. Credit: Jim Salter

Much of the way operating systems interact with storage devices revolves around finding ways to ask them to do as little as possible, since they are so frequently a system's biggest bottleneck. We talked about how reads are difficult to test, because the operating system does its best to cache them, and cache is orders of magnitude faster than storage. Operating systems like to "cheat" with writes just as much when given the chance for the same reason.

In the section on IOPS vs throughput above, we discussed the fact that most storage calls are limited not by how much data must be moved to or from disk, but by how many operations are necessary to fetch or store it. In order to accelerate writes, the operating system prefers to wait to commit them, rather than making each write the moment it's asked for. This standard behavior, called asynchronous writing, allows the operating system to order writes as nearly contiguous as possible.

Aggregating writes this way minimizes the number of IOPS required to write data to disk. Equally importantly, it allows later reads of the data to consume fewer IOPS—remember, the system can try to reorder writes, but for the most part it must read the data in whatever way it was written. A file stored in fragments now must also be fetched in fragments later. (Anyone who's done much bittorrenting has likely found this out the hard way.)

Sometimes, however, an application needs to know for certain that its writes are committed to storage *right now*, to prevent data loss or corruption in the event of a crash. To do so, the application calls `sync()` or `fsync()` after requesting the write. These calls force the operating system to flush any data in its write buffer—also known as "dirty" data—to disk immediately, no matter how inconvenient or inefficient that might be.

The worst-case scenario, from an IOPS perspective, is a 100 percent synchronous workload—meaning `sync()` is called after each individual block write. We test both fully synchronous and fully asynchronous workloads, using the `fio` arguments `fsync=1` for 100 percent synchronous, and `end_fsync=1` for 100 percent asynchronous.

Any `fio` write test *must* include `end_fsync=1` at a minimum, or the end result will be garbage. In 10 seconds, most systems will accept enough 4KiB asynchronous write requests to keep the disks chattering for minutes afterwards. Using `end_fsync=1`, forces `fio` to include the time spent actually committing the writes to disk in its throughput results.

Process number and iodepth

↑ *(left)* Bad news, friend—no RAID array is going to make a single-process random read significantly faster. Jim Salter

↑ *(right)* Eight processes and iodepth=8 allow big RAID arrays to shine. This is the only test you'll see both RAID5 and RAID10 performance actually scale linearly with number of disks. Jim Salter

Spoiler alert—RAID doesn't generally accelerate any one process. If your system isn't trying to do several things at once, you're unlikely to see any RAID configuration outperforming a single disk.

In order to demonstrate this fact, we ran each test with a single process and `iodepth=1` (meaning the system must process operations one at a time), and with eight parallel processes and `iodepth=8` (the system is allowed to stack up to eight operations in a queue to be processed).

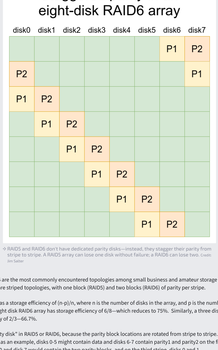
Understanding topology

Now that we understand how we're testing, and something about the theory involved, let's take a look at how our RAID arrays store data. There are three basic RAID topologies—mirrored, striped plain, and striped with parity.

We don't test striped plain topology—aka RAID0—because it's a load, clear invitation to data loss. Theoretically, a RAID0 array outperforms anything else—but in practice, a failed array has a performance of zero, and we refuse to contribute to anybody's decision to sacrifice real data at the RAID0 altar.

So, we tested four RAID protocols, falling into two families—striped with parity, and mirrored—and we'll take a brief look at each.

RAID5 and RAID6



↑ RAID5 and RAID6 don't have dedicated parity disks—instead, they stagger their parity from stripe to stripe. A RAID5 array can lose one disk without failure; a RAID6 can lose two. Credit: Jim Salter

RAID5 and RAID6 are the most commonly encountered topologies among small business and amateur storage admins' systems. These are striped topologies, with one block (RAID5) and two blocks (RAID6) of parity per stripe.

A striped array has a storage efficiency of (n-p)/n, where n is the number of disks in the array, and p is the number of parity blocks. So, an eight-disk RAID6 array has storage efficiency of 6/8—which reduces to 75%. Similarly, a three-disk RAID5 has a storage efficiency of 2/3—66.7%.

There is no "parity disk" in RAID5 or RAID6, because the parity block locations are rotated from stripe to stripe. Taking an eight disk RAID6 array as an example, disks 0-5 might contain data and disks 6-7 contain parity1 and parity2 on the first stripe. On the next stripe, disk 0 and disk 7 would contain the two parity blocks, and on the third stripe, disks 0 and 1.

When a disk fails, a striped parity array can reconstruct the data from the missing disk(s) so long as there is equivalent parity to reconstruct it with. This means that a RAID5 array can survive the loss of any one disk, but will fail if a second disk fails. RAID6 arrays, with two blocks of parity, can survive two disk failures. If you lose more disks than you have parity blocks, the array is toast, and all your data is gone.

RAID5 is a largely deprecated topology, considered unsafe due to the length of time it takes to rebuild a degraded array. Modern disks are many orders of magnitude larger than the disks current when striped RAID topologies were first designed—and while they have also gotten faster, speed hasn't kept up with capacity. It takes a lot longer to write a full drive worth of data now than it did ten years ago, or twenty.

A three-drive RAID5 array is probably fine, but a striped array any wider than that should use RAID6—or for higher-performance, give up entirely on striped-with-parity, and use RAID10 instead.

In our charts, you'll see "RAID6" results from three disks through eight disks, but the three disk data point is actually a RAID5 array. The four through eight disk results are all RAID6.

RAID1 and RAID10

RAID1 is the simplest possible topology: a mirror. In a RAID1 array, each block on each disk is identical to the block on the other disk in the array. Although RAID1 is most commonly seen in two disk arrays, it's possible to create a RAID1 with as many disks as you like—but no matter how many there are, each is identical, and the total capacity of the RAID1 array is that of a single disk.

RAID10 arrays are a stripe (without parity) of RAID1 arrays. So in an eight-disk RAID10 array, you have four RAID1 mirrors, and your writes are striped across them.

A RAID10 array will survive the loss of any single disk—and may survive further failures. The RAID10 array can lose any number of disks without failing, so long as each of its component mirrors has at least one functional disk remaining. But when any component mirror loses its last disk, the entire RAID10 goes down with it.

If we consider an eight disk RAID10 array, the odds of surviving one failed disk are 100%. The odds of surviving a second are 6/7—85.7%. The odds of surviving a third are 4/6—66.7%.

Of course, these odds are academic and rather naive. If you have two disks that came from the factory with iffy bearings, they're likely to fail in rapid succession. If both iffy drives were in the same mirror, your odds of surviving that second disk failure are much worse than the noty 85.7% that pure math gives us above.

The silver lining to this cloud is that RAID10 rebuilds are much faster than RAID5 and RAID6 rebuilds, so your odds of surviving until the new disk finishes are correspondingly better than surviving through a much longer rebuild window under RAID5 or RAID6.

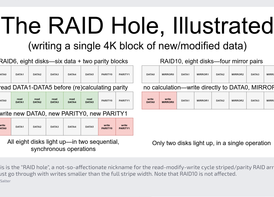
In our performance charts, we show a line from two disks through eight disks for RAID10. The first datapoint, for two disks, is obviously a simple RAID1. The datapoints for four, six, and eight disks are RAID10. We draw a line through all points for ease of interpretation—but there is no three, five, or seven disk RAID10 in the actual data, for hopefully obvious reasons.

Performance

Finally, all that pesky theory is behind us, and it's (finally!) time to get to the real numbers. *Phew.* Thanks for sticking with us this far.

We're going to break our tests down into two major sections: single process and eight process. Each of those major sections will be further broken down into 4KiB blocksize and 1MiB blocksize results.

Single process, iodepth=1



↑ RAID arrays don't accelerate single-process operations. If you aren't stacking up a bunch of ops in parallel, adding more disks in parallel doesn't help. Jim Salter

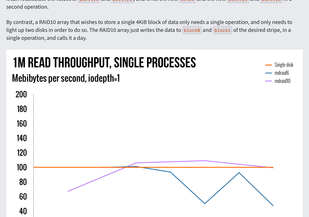
↑ *(left)* We can see a tiny advantage for larger RAID10 arrays with single process sync writes. It's nothing to write home about, though, and RAID6 just stinks here. Jim Salter

↑ *(right)* We begin to see some serious acceleration for big RAID10 arrays with single-process asynchronous writes. RAID6, again, stinks—and gets worse the wider the array is. Jim Salter

We spoiled this surprise up top, but here it is again—RAID arrays don't really accelerate 4KiB single-process operations. If you've only got a single process accessing your storage, it won't typically get much faster than a single disk—and much of the time, it'll be significantly slower. In particular, we see asynchronous performance on RAID6 arrays dive straight into the toilet, getting worse the wider the array is.

The exception here is asynchronous 4KiB writes on RAID10. This is because, technically, asynchronous writes aren't really single process at all—the operating system, after aggregating a number of write operations in RAM, can commit them to disk in parallel. It's free to do this in the most efficient order possible, so performance tends to increase in proportion to the number of disks.

We don't see the same effect with 4KiB reads, because the operating system can't do as much to optimize the process—it's still free to issue operations in parallel, but it doesn't have the luxury of picking them so that one is on each disk of the array each time, and it can make full-stripe operations. Instead, it's forced to read them from whatever they happen to be, in roughly the order the application requests them.



↑ This is the "RAID hole", a not-so-affectionate nickname for the read-modify-write cycle striped/parity RAID arrays must go through with writes smaller than the full stripe width. Note that RAID10 is not affected. Jim Salter

↑ *(left)* This RAID6 array is committing 4K writes to disk—the workload itself is pure write, as we can see in /dev/nvme7n-row, but the individual disks spend as many operations reading as they do writing, leading to crappy performance. Jim Salter

↑ *(right)* When we issue the same pure 4K write workload to a RAID10 array, we see far higher performance—all IOPS are write, and there are many more of them as well. Jim Salter

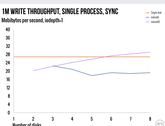
We suspect a lot of readers will be surprised by RAIDz's performance getting significantly worse as the number of disks gets larger. This is due to a phenomenon not-so-affectionately referred to as the "RAID hole."

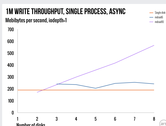
The RAID hole refers to the read-modify-write cycle striped/parity arrays must go through in order to commit any write smaller than the full stripe width. The values of parity blocks must be calculated from the data blocks of all disks of the stripe—so to write a single 4KiB block to a RAID6 array, the system must light up all of its disks—and it must do so in two consecutive, binding operations.

Let's examine a hypothetical, eight disk RAID6 array. We'll refer to the eight blocks of one stripe as `datad0` through `datad7`, `paritydd`, and `parityd`. The system wants to store a new value in `datad0`—but first, it must read `datad1`, `datad2`, `datad3`, `datad4`, and `datad5`. Now that the system knows the values of (new) `datad0` and (unchanged) `datad1` through `datad5`, it can recalculate the values of `paritydd` and `parityd`, and write the new `datad0` and the new `paritydd` and `parityd` in a second operation.

By contrast, a RAID10 array that wishes to store a single 4KiB block of data only needs a single operation, and only needs to light up two disks in order to do so. The RAID10 array just writes the data to `block0` and `block1` of the desired stripe, in a single operation, and calls it a day.

→ Even with large blocksizes, single process reads aren't significantly accelerated on RAID arrays. Performance won't get much better than single-disk, at best—and it can be significantly worse.
Jim Salter





→ *(left)* You should be picking up a pattern, here. The throughput numbers are bigger for 1MiB ops than for 4KiB ops, but the shape of the graphs is roughly the same. Still no acceleration for single-process ops.
Jim Salter

→ *(right)* 1MiB async writes are a lot like 4KiB async writes—we see RAID6 bobbing along at or slightly above single disk performance, while larger RAID10 arrays take off running.
Jim Salter

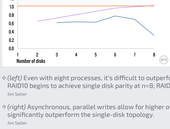
When we move from 4KiB to 1MiB blocksizes, single process operation throughput goes up, but the overall shapes of the graphs remain roughly constant. RAID10 beats RAID6 at every level, but we only significantly exceed single disk performance with RAID10 arrays doing asynchronous writes.

RAID5/RAID6 arrays also generally manage to outperform a single disk when doing 1MiB asynchronous writes, but not by much—and the effect doesn't scale up with the number of disks.

In fact, RAID6 performance will eventually trend sharply down as the array size increases—but we don't see that happening yet; the single process and low iodepth here is actually saving RAID6 from the worst of its own follies.

Eight processes, iodepth=8

→ With the greater parallelism offered by eight simultaneous processes and iodepth=8, we begin to see acceleration generally trending upwards with larger disk count.
Jim Salter



→ *(left)* Even with eight processes, it's difficult to outperform a single disk when doing 100% synchronous writes. RAID10 begins to achieve single disk parity at n=8; RAID5 drops into the abyss at the same level.
Jim Salter

→ *(right)* Asynchronous, parallel writes allow for higher overall throughput—and they also allow RAID10 to significantly outperform the single-disk topology.
Jim Salter

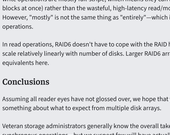
Servers typically spend more time servicing parallel operations than single ones. This is a good thing for those with RAID, since RAID performance scales up much better with parallel operations than single ones.

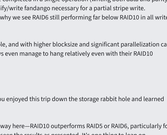
The sharp peak in RAID6 performance at n=4 is an accidental correlation of the stripe width to the workload—we're getting lucky with how often a full stripe read provides us with two of the blocks we're looking for instead of one, and we're unable to "waste" more than one disk's IOPS by getting a block we don't actually want with that full stripe read. We see a similar effect with the two disk RAID1 mirror; otherwise, both topologies are basically flat at roughly double the performance of a single disk.

Moving on to write throughput, we once again see larger RAID10 arrays slowly achieving parity with single-disk performance, and larger RAID6 arrays enthusiastically diving into the toilet due to the RAID hole. When we allow writes to commit asynchronously, we see RAID6 achieve roughly 1/3 of single-disk performance and slowly trend downward, while RAID10 climbs to roughly double single-disk performance and essentially plateaus there.

It's worth noting that although the shape of these graphs is quite similar to the shape of the single-process 4KiB graphs, the scale is different—asynchronous writes are achieving roughly triple the throughput, across the board, with the higher parallelism here.

→ RAID6 finally manages to keep up—mostly—with RAID10 on 1MiB parallel reads. Note that the effect begins tapering off at n=6. It'll likely flatten almost entirely between n=8 and n=10 (not tested).
Jim Salter





→ *(left)* With eight-process 1MiB sync writes, RAID6 performance once again falls sharply compared to RAID10. It does at least manage to keep up with a single disk.
Jim Salter

→ *(right)* RAID10 once again scales like a champion with number of disks. RAID6 does nowhere near as well, but manages a significant peak above single-disk at n=7.
Jim Salter

If you've been waiting for the "OMG, big numbers!" portion of our testing, congratulations—you've arrived. Significantly parallel large blocksize operations are the workload that's friendliest to RAID performance, and we see all arrays at least hanging with—and typically exceeding—single disk performance here.

This test category is, by far, the most favorable to striped/parity arrays. RAID6, for the first time, never drops significantly below single-disk performance levels—and significantly exceeds them, in both reads and (for wider RAID6 arrays) asynchronous writes.

The effect of the RAID hole on RAID6 writes is present, but relatively minimal, in these larger blocksize operations. The 1MiB write operations are mostly full-stripe, which means they can be completed in a single operation (writing both data and parity blocks at once) rather than the wasteful, high-latency read/modify/write fandango necessary for a partial stripe write. However, "mostly" is not the same thing as "entirely"—which is why we see RAID6 still performing far below RAID10 in all write operations.

In read operations, RAID6 doesn't have to cope with the RAID hole, and with higher blocksize and significant parallelization can scale relatively linearly with number of disks. Larger RAID6 arrays even manage to hang relatively even with their RAID10 equivalents here.

Conclusions

Assuming all reader eyes have not glossed over, we hope that you enjoyed this trip down the storage rabbit hole and learned something about what to expect from multiple disk arrays.

Veteran storage administrators generally know the overall takeaway here—RAID10 outperforms RAID5 or RAID6, particularly for synchronous operations—but we suspect few will have actually *seen* the results as presented. It's one thing to lean on theoretical rules of thumb; it's another thing entirely to empirically test and graph results.

We suspect that even many storage greybeards will be surprised at just how thoroughly RAID10 outperforms RAID6 across the board and without exception. Some may also be surprised at just how difficult it is to outperform—or even hold even with—single disk performance for the most challenging workloads, no matter what your topology.

After all this, we have several take-away recommendations borne out by our tests:

- RAID10 for performance—including, but *not* limited to database performance. Yes, fileservers too!
- **RAID is not a backup.** The parity/redundancy in the array mitigates downtime, *not* catastrophic failure. Back up your arrays!
- Be extremely wary of RAID5—a single block of parity isn't sufficient for protection for any but the very smallest arrays.
- RAID is not a backup. **The parity/redundancy in the array mitigates downtime, not catastrophic failure.** Back up your arrays!
- At four disks wide, RAID6 vs RAID10 is a difficult decision—RAID6's performance hasn't dived into the toilet yet, and it can survive more guaranteed failures than RAID10. Choose wisely!
- RAID is not a backup. The parity/redundancy in an array mitigates downtime, *not* catastrophic failure. **Back up your array!**
- Wide RAID6 arrays are, for almost all realistic workloads, absolute hot garbage when it comes to performance. Be careful here—you get more effective storage than RAID10, but it may not be worth it.
- Have we asked you to back up your RAID array yet? Please don't neglect your backups...

If this exercise got your sysadmin brains lighting up, don't fret. In the future, we plan to follow up with similar performance trend testing on ZFS. Because while ZFS topologies are frequently (and erroneously) equated directly with standard RAID, in reality those are considerably more complex.

RELATED STORIES

I updated my crusty old Pentium G based server—the results are worth sharing
A Ryzen CPU and a Roswell case got us big gains at about the same cost.

How fast are your disks? Find out the open source way, with fio
The most reliable way to test disks is down-and-dirty, on the command line.

Listing image: Jim Salter

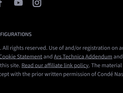
JIM SALTER FORMER TECHNOLOGY REPORTER
Jim is an author, podcaster, mercenary sysadmin, coder, and father of three—not necessarily in that order.

318 COMMENTS

PREV STORY

NEXT STORY

MOST READ


1. Teen creates memecoin, dumps it, earns $50,000
2. After critics decry Orion heat shield decision, NASA reviewer says agency is correct
3. Booking.com says typos giving strangers access to private trip info is not a bug
4. Lower-cost sodium-ion batteries are finally having their moment
5. Amazon secretly slowed deliveries, deceived anyone who complained, lawsuit says

ars TECHNICA

Ars Technica has been separating the signal from the noise for over 25 years. With our unique combination of technical savvy and wide-ranging interest in the technological arts and sciences, Ars is the trusted source in a sea of information. After all, you don't need to know everything, only what's important.

MORE FROM ARS
ABOUT US
STAFF DIRECTORY
NEWSLETTERS
ARS VIDEOS
GENERAL FAQ
RSS FEEDS

CONTACT
CONTACT US
ADVERTISE WITH US
REPRINTS

PRIVACY CONFIGURATIONS

© 2024 Condé Nast. All rights reserved. Use of and/or registration on any portion of this site constitutes acceptance of our User Agreement and Privacy Policy and Cookie Statement and Ars Technica Addendum and Your California Privacy Rights. Ars Technica may earn compensation on sales from links on this site. Read our affiliate link policy. The material on this site may not be reproduced, distributed, transmitted, cached or otherwise used, except with the prior written permission of Condé Nast. Ad Choices