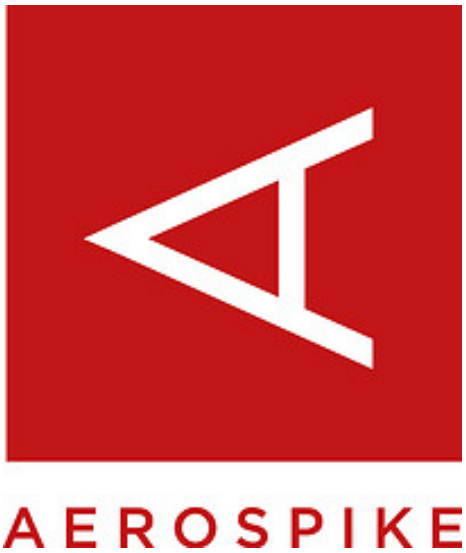
Russ' 10 Ingredient Recipe for Making 1 Million TPS on $5K Hardware

Monday, September 10, 2012 at 9:20AM

Todd Hoff in Product, Strategy



My name is Russell Sullivan, I am the author of AlchemyDB: a highly flexible NoSQL/SQL/DocumentStore/GraphDB-datastore built on top of redis. I have spent the last several years trying to find a way to sanely house multiple datastore-genres under one roof while (almost paradoxically) pushing performance to its limits.

I recently joined the NoSQL company Aerospike (formerly Citrusleaf) with the goal of incrementally grafting AlchemyDB's flexible data-modeling capabilities onto Aerospike's high-velocity horizontally-scalable key-value data-fabric. We recently completed a peak-performance TPS optimization project: starting at 200K TPS, pushing to the recent community edition launch at 500K TPS, and finally arriving at our 2012 goal: **1M TPS on $5K hardware**.

Getting to one million over-the-wire client-server database-requests per-

second on a single machine costing $5K is a balance between trimming overhead on many axes and using a shared nothing architecture to isolate the paths taken by unique requests.

Even if you aren't building a database server the techniques described in this post might be interesting as they are not database server specific. They could be applied to a ftp server, a static web server, and even to a dynamic web server.

Here is my personal recipe for getting to this TPS per dollar.

The Hardware

Hardware is important, but pretty cheap at 200 TPS per dollar spent:

1. Dual Socket Intel motherboard
2. 2*Intel X5690 Hexacore @3.47GHz
3. 32GB DRAM 1333
4. 2 NIC ports of an Intel quad-port NIC (each NIC has 8 queues)

Select the Right Ingredients

The architecture/software/OS ingredients used in order to get optimal peak-performance rely on the combination and tweaking of ALL of the ingredients to hit the sweet spot and achieve a VERY stable 1M database-read-requests per-second over-the-wire.

It is difficult to quantify the importance of each ingredient, but in general they are in order of descending importance.

Select the Right Architecture

First, it is imperative to start out with the right architecture, both vertical

and horizontal scalability (which are essential for peak-performance on modern hardware) flow directly from architectural decisions:

1. **100% shared nothing architecture**. This is what allows you to parallelize/isolate. Without this, you are eventually screwed when it comes to scaling.

2. **100% in-memory workload**. Don't even think about hitting disk for 0.0001% of these requests. SSDs are better than HDDs, but nothing beats DRAM for the dollar for this type of workload.

3. **Data lookups should be dead-simple**, i.e.:

1. Get packet from event loop (event-driven)
2. Parse action
3. Lookup data in memory (this is fast enough to happen in-thread)
4. Form response packet
5. Send packet back via non-blocking call

4. **Data-Isolation**. The previous lookup is lockless and requires no hand-off from thread-to-thread: this is where a shared-nothing architecture helps you out. You can determine which core on which machine a piece of data will be written-to/served-from and the client can map a tcp-port to this core and all lookups go straight to the data. The operating system will provide the multi-threading & concurrency for your system.

Select the Right OS, Programming Language, and Libraries

Next, make sure your operating system, programming language, and libraries are the ones proven to perform:

5. **Modern Linux kernel**. Anything less than CentOS 6.3 (kernel 2.6.32) has serious problems w/ software interrupts. This is also the space where we can expect a 2X improvement in the near future; the Linux kernel is currently being upgraded to improve multi-core efficiency.

6. **The C language**. Java may be fast, but not as fast as C, and more importantly: Java is less in your control and control is the only path to peak performance. The unknowns of garbage collection frustrate any and all attempts to attain peak performance.

7. **Epoll**. Event-driven/non-blocking I/O, single threaded event loop for high-speed code paths.

Tweak and Taste Until Everything is Just Right

Finally, use the features of the system you have designed. Tweak the Hardware & OS to **isolate** performance critical paths:

8. **Thread-Core-Pinning**. Event loop threads reading and writing tcp packets should each be pinned to their own core and no other threads should be allowed on these cores. These threads are so critical to performance; any context switching on their designated cores will degrade peak-performance significantly.

9. **IRQ affinity from the NIC**. To avoid ALL soft interrupts (generated by tcp packets) bottlenecking on a single core. There are different methodologies depending on the number of cores you have:

1. **For QuadCore CPUs**: round-robin spread IRQ affinity (of the NIC's Queue's) to the Network-facing-event-loop-threads (e.g. 8 Queue's, map 2 Queue's to each core)

2. **On Hexacore (and greater) CPUs**: reserve 1+ cores to do nothing but IRQ-processing (i.e. send IRQ's to these cores and don't let any other thread run on these cores) and use ALL other cores for Network-facing-event-loop-threads (similarly running w/o competition on their own designated core). The core receiving the IRQ will then signal the recipient core and the packet has a near 100% chance of being in L3 cache, so the transport of the packet from core to core is near optimal.

10. **CPU-Socket-Isolation via PhysicalNIC/PhysicalCPU pairing**. Multiple CPU sockets holding multiple CPUs should be used like multiple machines. Avoid inter-CPU communication; it is dog-slow when compared to communication between cores on the same CPU die. Pairing a physical NIC port to a PhysicalCPU is a simple means to attain this goal and can be achieved in 2 steps:

1. Use IRQ affinity from this physical NIC port to the cores on its designated PhysicalCPU
2. Configure IP routing on each physical NIC port (interface) so packets are sent from its designated CPU back to the same interface (instead of to the default interface)

This technique isolates CPU/NIC pairs; when the client respects this, a Dual-CPU-socket machine works like 2 single-CPU-socket machines (at a much lower TCO).

That is it. The 10 ingredients are fairly straightforward, but putting them all together, and making your system really hum, turns out to be a pretty difficult balancing act in practice. The basic philosophy is to isolate on all axis.

The Proof is Always in the Pudding

Any 10 step recipe is best illustrated via an example: the client knows (via multiple hashings) that dataX is presently on core8 of ipY, which has a predefined mapping of going to ipY:portZ.

The connection from the client to ipY:portZ has previously been created, the request goes from the client to ipY:(NIC2):portZ.

- NIC2 sends all of its IRQs to CPU2, where the packet gets to core8 w/ minimal hardware/OS overhead.
- The packet creates an event, which triggers a dedicated thread that runs w/o competition on core8.
- The packet is parsed; the operation is to look up dataX, which will be in its local NUMA memory pool.
- DataX is retrieved from local memory, which is a fast enough operation to not benefit from context switching.
- The thread then replies with a non-blocking packet that goes back thru only cores on the local CPU2, which sends ALL of its IRQs to NIC2.

Everything is isolated and nothing collides (e.g. w/ NIC1/CPU1). Software interrupts are handled locally on a CPU. IRQ affinity insures software interrupts don’t bottleneck on a single core and that they come from and go from/to their designated NIC. Core-to-core communication happens ONLY withIN the CPU die. There are no unnecessary context switches on performance-critical code paths. TCP packets are processed as events by a single thread running dedicated on its own core. Data is looked up in the local memory pool. This isolated path is the closest software path to what actually physically happens in a computer and the key to attaining peak performance.

At Aerospike, **I knew I had it right** when I watched the output of the "top" command, (viewing all cores) and there was near zero idle % cpu and also a very uniform balance across cores. Each core had exactly the same signature, something like: **us%39 sy%35 id%0 wa%0 si%22**.

Which is to say software-interrupts from tcp packets were using 22% of the core, context switches passing tcp-packets back and forth from the operating system were taking up 35%, and our software was taking up 39% to do the database transaction.

When the perfect balance across cores was achieved optimal performance was achieved, from an architectural standpoint. We can still streamline our software but at least the flow of packets to & fro Aerospike is near optimal.

Data is Served

Those are my 10 ingredients that got Aerospike's server to one million over-the-wire database requests on a $5K commodity machine. Mixed correctly, they not only give you incredible raw speed, they give you stability/predictability/over-provisioning-for-spikes at lower speeds. Enjoy •

Related Articles

- Big List Of 20 Common Bottlenecks
- AeroSpike, the former Citrusleaf

Article originally appeared on High Scalability (http://highscalability.com/).

See website for complete article licensing information.