Musings

RIK FARROW

Rik is the editor of *;login:*.
rik@usenix.org

I just got back from NSDI in Boston. Wintry weather (where's spring?) and interesting talks, as usual. I really liked the LEET talks [1] about spam and botnets. The scale of these operations is just amazing. It is also amazing to learn just what can be done with a handful of servers, even if they are working to send out over a trillion spam emails. The NSDI talks also covered issues of scalability, with one of the best papers describing how GPUs can be used to support parallel SSL operations.

Scalability of today's systems remains an intriguing question, one that Konstantin Shvachko takes on in this issue. Konstantin has worked on the largest known Hadoop cluster, at Yahoo!, and is now working to build Hadoop clusters at eBay. In his article, he examines the limits to scalability.

The Factory Model

I am always fascinated—like many computer enthusiasts, I imagine—by the latest offerings of Intel and AMD. CPU clock speeds have reached approximate upper limits, and there are real reasons for this. But it is hard to think clearly about these limits, when processors and I/O busses are just so darn fast. Clock speeds in gigahertz, where one gigahertz means a cycle time of one nanosecond, are far afield from the human sensory realm. So I decided to think about a factory model instead.

My imaginary factory has an assembly line that can crank out 3.4 thousand widgets per hour—as long as you keep it supplied with parts. Just to get it to start producing finished widgets, you have to feed it the parts it needs and keep the production pipeline as full as you can. And there's a problem with my factory: the methods for keeping the assembly line fed with parts is nowhere near as fast as the assembly line itself.

The assembly line is supplied by a hierarchy of specialized feeds. Those next to the assembly line work three times slower than the assembly line itself—and this feed device is often missing the desired parts. The next level in the hierarchy is 10 times slower than the assembly line, and while it is much bigger than the first-level feed, it too may often not contain the needed parts.

The level beyond resembles a just-in-time warehouse, with huge arrays of shelves and automated devices that bring the desired parts to the assembly line feeds. But this huge warehouse can be over 100 times slower. Sometimes the parts required can be delivered in batches, and getting the first batch of parts is only 60 times slower than the assembly line. Additional parts from the same part of the

warehouse can be delivered twice as fast, but that is, at best, still half the speed of the second-level feed. And if the parts required come from more random parts of the warehouse, getting each part is always 60 times slower.

If our modern, just-in-time warehousing fails, and the factory needs a part that is not there, it is going to take thousands of times longer to arrive in the warehouse (before it can be delivered to the feeders for the assembly line). Thus, even though our assembly line can crank out 3.4 thousand widgets per hour, it will never actually do so. It can only produce widgets as fast as the slowest part of the part feeding system. Hopefully, the parts will always be in the warehouse, and the assembly line can churn out perhaps 600 widgets per hour.

Oh, I forgot to mention that we actually have four identical assembly lines all sharing the upper-level feed and warehouse supplies, so their total production has the same limitations as the single assembly line.

Allegorically Speaking

I modeled my factory on an Intel Sandy Bridge CPU, using the LGA 1155 socket [2]. This CPU sports clock speeds from 1.4 to 3.4 GHz. While the Intel spec sheet says it can perform up to 2.5 billion memory transfers per second, that is best case because of how DRAM works. Actually, this is sort of like saying you have a car that can go 200 miles per hour, but only for a second at a time. The rest of the time it cruises at 65 mph.

The elaborate "feed" system in my factory represents the memory hierarchy: registers, L1 and L3 caches, DRAM, and, finally, the dreadfully slow (by comparison) hard disk and network. Only registers can provide data at CPU clock speeds. Fetching data from L1 cache will take three clock cycles, and from L3 (which is up to 100 times larger than L1) takes many more clock cycles. It is not that the memory in cache is slower than register memory: both are built of static RAM (SRAM). It just takes longer to discover if the data you need is in a particular cache, and the larger the cache, the longer it takes to search for a hit in the cache.

DRAM is different from SRAM. SRAM requires five transistors to hold one bit of memory, but DRAM only requires one transistor and one capacitor. DRAM both has a smaller footprint and requires less energy than SRAM, making it suitable for bulk storage (the warehouse in my analogy). But it is much slower than SRAM for several reasons, including how DRAM is addressed: the address is supplied in two parts, the row, then the column, each with multiple nanoseconds of latency. Then the actual data must be "sensed" before it can be read. So reading data from a random part of our memory warehouse can take up to tens of nanoseconds. In "burst mode" (our car that can go really fast for a short time), as long as the row address remains unchanged, DRAM can provide 8 or 16 bytes every 10 nanoseconds. Once the row address is changed, there is an additional waiting period before burst mode can start again. And if random access is really random, forget about burst mode (and the 200 mph car).

Our warehouse of DRAM gets supplied by disk or network. Practically speaking (that is, no 200 mph car), each disk can supply 60 MB/s, and a one gigabit network can supply 119 MB/s. When you compare these data rates to DRAM data rates, they are many times slower. On top of that, disk and network latency is measured in milliseconds, that is, in millions of nanoseconds. For a CPU clocked at 3.4 GHz,

about .29 nanoseconds per clock tick, a latency of 10 milliseconds translates into 340,000 clock cycles. That's a very long time to wait when you are a CPU.

As a friend from Intel recently told me, it's all about memory. If you can provide enough memory bandwidth to keep the CPU busy, you are doing great. The best rate of speed for the memory controller used in our example is 21.3 GB/s. But if our memory yields 16 bytes every 10 nanoseconds, even for a short amount of time (one page), that's 1.6 GB/s. Ooops. Even if I am off by a factor of two, that is still far below the maximum rate of the memory controller, and certainly far below the rate of registers and cache, especially when you consider that there may be four cores of CPU, each waiting for four or eight bytes of data.

Before you decide not to buy that really fast CPU, keep in mind that I am making a point through exaggeration here. CPUs can process more data than memory can provide, especially fast, manycore processors. But many applications don't require reading and writing vast amounts of random data, or even better, they feature locality of both code and data. These applications do perform better on faster CPUs. Also, the threading provided in modern processors also helps hide latency by allowing the processors to work on something else instead of idly waiting for data. So things are not as bad as I made them seem.

The memory hierarchy is real and is the biggest challenge facing hardware, compiler, and system designers and anyone who wants to tune an application so that it performs as fast as possible. Otherwise, you have idle assembly lines, that is, CPUs.

Clusters and Clouds

Hadoop provides a mechanism for dealing with the huge disparity between data and processor speed when you want to process massive amounts of data. First, you focus on having the data you want to process local to the processor by dividing that huge amount of data into many slices stored on many disks. Then you process those slices in parallel. Divide and conquer, the way forward in the days of Caesar, and still useful today with big data.

Konstantin brings us up to date on scalability issues with Hadoop. Hadoop has become increasingly popular as a method for processing enormous amounts of data. It is still not the fastest way—for that, you need more advanced methods, such as Google's Percolator [4]. Konstantin brings a great amount of experience to the table, as he has worked with Hadoop and its data storage systems for many years.

Jeff Darcy describes a project he has been working on, CloudFS. Jeff provides a very thoughtful description of the challenges facing any cloud file system. Then he explains how he deals with these problems by leveraging GlusterFS to create CloudFS, both open source projects. I first heard about CloudFS during a FUDCon, where Jeff spoke just as eloquently about the issues facing scalable, secure, manageable, and reliable cloud storage.

I met Steve Hetzler after the tutorial he presented at FAST. Steve is an IBM Fellow, and not one to mince words. And Steve doesn't like how disk error rates are presented by storage companies. After all, how many of you read bits from your hard drives? Last time I checked, your only option was to read sectors, and you could only do that at a certain maximum rate, based on IOPS.

Steve also casts his steely gaze upon solid-state devices (SSDs). While flash appears to solve many issues for I/O bound applications, its endurance is an open question and, according to Steve, not one we can answer any time soon.

Stuart Kendrick volunteered to write about the hazards of high availability (HA). Stuart has worked for many years on servers designed to be highly available but which turned out to be highly unavailable (HU) instead. He shares his experience in testing HA in the hopes that you too might have HA instead of HU.

David Blank-Edelman dances around an elegant module for serving up HTTP content. Two years ago, David wrote about the CGI::Application framework [5]. This time, David introduces Dancer, another framework that strives to keep things simple. Dancer is a port of Sinatra (who is not a dancer), a Ruby framework that is simpler than Rails. Just the thing if you know Perl and need something like Rails, but simpler.

Dave Josephson dazzles us with a handful of monitoring gems, but not until he has had a chance to rant. And what a wonderful rant (unless you or your significant other is an auditor). Dave shares his hard-won experience in monitoring distributed, Internet-facing servers with four great monitoring tools.

Robert Ferrell brings a (literally) fevered imagination to the subject of viruses.

Peter Galvin decided not to write for this issue. I expect he will be back by August, but who can tell what the future may bring?

We close out this issue with summaries from FAST '11 and NSDI '11. Having taken a peek at the USENIX Web logs, I can tell you that the FAST paper on proximal writes by several NetApps employees is extraordinarily popular. I personally learned a lot about deduplication, working with flash, dealing with very large directories, and more by attending FAST and reading the summaries. At NSDI, there were no invited talks. I liked the CIEL paper (no surprise there) and the SSLShader paper, where researchers describe creating an SSL accelerator with an older GPU. The performance limitation for SSLShader is not the GPU; the memory bandwidth available is the bottleneck. What a surprise (not). As always, you can download any paper from FAST or NSDI, as well as watch videos of presentations or view the slides [6].

I don't want to denigrate the performance improvements we have seen in processor and system design. I mention Intel Core i5 because I am using one in my new desktop. Not only does it provide me with a vast increase in CPU performance, I finally have decent graphics performance as well. Not that I really have any great need for either, to be honest. My 8 MHz 68k-based System V server kept up with my typing just fine. But sometimes it is nice to play with Google Earth. I've found that looking at maps is one thing, but flying along over familiar routes grants me an entire new perspective on how the world around me appears in reality. Or at least, something closer to reality than my own mental maps.

Designing balanced systems is still the way forward. We now have incredibly powerful systems, even if their processors far outpace the I/O that feeds them. The path forward includes new systems and software designs that play to the strengths of the hardware we have, especially through designs that emphasize the most efficient memory footprint possible or, at least, as much sequential I/O as possible and clusters of systems for dealing with Big Data.

References

[1] 4th USENIX Workshop on Large-Scale Exploits and Emergent Threats: http://www.usenix.org/events/leet11/tech/.

[2] Intel Core i5 CPU: http://ark.intel.com/Product.aspx?id=47341.

[3] Intel Core i5-750 and i7-870 processors, The Tech Report: http://techreport.com/articles.x/17545/5.

[4] Daniel Peng and Frank Dabek, "Large-Scale Incremental Processing Using Distributed Transactions and Notifications," OSDI '10: http://www.usenix.org/events/osdi10/tech/full_papers/Peng.pdf.

[5] David N. Blank-Edelman, "Practical Perl Tools: Scratch the Webapp Itch with CGI::Application, Part 1," *;login:,* vol. 34, no. 4, August 2009: http://www.usenix.org/publications/login/2009-08/pdfs/blank-edelman.pdf.

[6] 9th USENIX Conference on File and Storage Technologies (FAST '11), Technical Sessions: http://www.usenix.org/events/fast11/tech/.