Know Your Algorithms

STOP USING HARDWARE TO SOLVE SOFTWARE PROBLEMS.

who is KV?

click for video

Dear KV,

My team and I are selecting a new server platform for our project and trying to decide whether we need more cores or higher-frequency CPUs, which seems to be the main tradeoff to make on current server systems. Our system is deployed on the highest-end systems and, therefore, the highest-frequency systems we could buy two years ago. We run these systems at 100 percent CPU utilization at all times. Our deployment does not consume a lot of memory, just a lot of CPU cycles, and so we're again leaning toward buying the latest, top-of-the-line servers from our vendor. We've looked at refactoring some of our software, but from a cost perspective, expensive servers are cheaper than expensive programmer time, which is being used to add new features, rather than reworking old code. In your opinion, what is more important in modern systems: frequency or core count?

Richly Served

Dear Served,

I really wish I knew who your vendor is, so I could get a cut of this incredibly lucrative pie. As the highest-end servers currently enjoy a massive markup, your salesperson probably has a biological mishap every time you call.

The short answer to your question about frequency vs. core count is, "You tell me." It seems as if you've

spent little or no time trying to understand your own workload and have simply fallen for the modern fallacy of "newer will make it better." Even apart from the end of frequency scaling, it has rarely been the case that just adding more oomph to a system is the best way to improve performance. The true keys to improving performance are measurement and an understanding of algorithms.

Knowing that your CPU is in use 100 percent of the time doesn't tell you much about the overall system other than it's busy, but busy with what? Maybe it's sitting in a tight loop, or some clown added a bunch of delay loops during testing that are no longer necessary. Until you profile your system, you have no idea why the CPU is busy. All systems provide some form of profiling so that you can track down where the bottlenecks are, and it's your responsibility to apply these tools before you spend money on brand new hardware—especially given how wacky new hardware has been in the past five years, particularly as a result of NUMA (non-uniform memory access) and all the crazy security mitigations that have sapped the life out of modern systems to deal with Spectre and the like. There are days when KV longs for the simplicity of a slow, eight-bit microprocessor, something one could understand by looking at the stream of assembly flying by. But those days are over, and, honestly, no one wants to look at cats on a Commodore 64, so it's just not a workable solution for the modern Internet.

Since I've talked about measurement before, let's talk now about the importance of algorithms. Algorithms are at the heart of what we as software engineers do, even though this fact is often hidden from us by libraries and

The simplest way to start thinking about your algorithm is the number of operations required per unit of input.

well-traveled APIs. The theory, it seems, is that hiding algorithmic complexity from programmers can make them more productive. If I can stack boxes on top of boxes—like little Lego bricks—to get my job done, then I don't need to understand what's inside the boxes, only how to hook them together. The box-stacking model breaks down when one or more of the boxes turns out to be your bottleneck. Then you'll have to open the box and understand what's inside, which, hopefully, doesn't look like poisonous black goo.

A nuanced understanding of algorithms takes many years, but there are good references, such as Donald Knuth's series, *The Art of Computer Programming,* which can help you along the way. The simplest way to start thinking about your algorithm is the number of operations required per unit of input. In undergraduate computer science, this is often taught by comparing searching and sorting algorithms. Imagine that you want to find a piece of data in an array. You know the value you want to find but not where to find it. A naive first approach would be to start from element 0 and then compare your target value to each of the elements in turn. In the best case, your target value is present at element 0, in which case you've executed a very small number of instructions, perhaps only one or two. The worst-case scenario is that your target element does not exist at all in the array and you will have executed many instructions—one comparison for every element of the array—only to find that the answer to your search is empty. This is called a linear search.

For many data structures and algorithms, we want to know the best, worst, and average number of operations it takes to achieve our goal. For searching an array,

best is 1, worst is N (the size of the array), and average is somewhere in the middle. If the data you are storing and searching is very small—a few kilobytes—then an array is likely your best choice. This is because even the worst-case search time is only a few thousand operations, and on any modern processor, that's not going to take a long time. Also, arrays are very simple to work with and understand. It is only when the size of the data set grows into megabytes or larger that it makes sense to pick an algorithm that, while it might be more complex, is able to provide a better average number of operations.

One example might be to pick a hash table that has an average search time of one operation and a worst search time of N—again the number of elements in the table. Hash tables are more complex to implement than arrays, but that complexity may be worth the shorter search time if, indeed, searching is what your system does most often. There are data structures and search algorithms that have been developed over the past 30 years with varying performance characteristics, and the list is too long, tedious, and boring to address in depth here. The main considerations are how long does it take, in the best, worst, and average cases, to

1. Add an element to the data structure (insertion time),
2. Remove an element,
3. Find an element.

Personally, I never bother with the best case, because I always expect that, on average, everything will be worst case. If you're lucky, there is already a good implementation of the data structure and algorithm you need in a different box from the one you're using now, and

Related articles

- KV the Loudmouth
  To buy or to build, that is the question.
  https://queue.acm.org/detail.cfm?id=1255426
- 10 Optimizations on Linear Search
  The operations side of the story
  Thomas A. Limoncelli
  https://queue.acm.org/detail.cfm?id=2984631
- Computing without Processors
  Heterogeneous systems allow us to target our programming to the appropriate environment.
  Satnam Singh
  https://queue.acm.org/detail.cfm?id=2000516

instead of having to open the box and see the goo, you can choose the better box and move on to the next bottleneck. No matter what you're doing when optimizing code, better choice of algorithms nearly always trumps higher frequency or core count.

In the end, it comes back to measurement driving algorithm selection, followed by more measurement, followed by more refinement. Or you can just open your wallet and keep paying for supposedly faster hardware that never delivers what you think you paid for. If you go the latter route, please contact KV so we can set up a vendor relationship, which will go directly to pay my bar tab.

KV

Kode Vicious, *known to mere mortals as George V. Neville-Neil, works on networking and operating-system code for fun and profit. He also teaches courses on various subjects related to programming. His areas of interest are code spelunking, operating systems, and rewriting your bad code (OK, maybe not that last one). He earned his bachelor's degree in computer science at Northeastern University in Boston, Massachusetts, and is a member of ACM, the Usenix Association, and IEEE. Neville-Neil is the co-author with*

Marshall Kirk McKusick and Robert N. M. Watson of The Design and Implementation of the FreeBSD Operating System *(second edition). He is an avid bicyclist and traveler who currently lives in New York City.*