Brawny cores still beat wimpy cores, most of the time

Urs Hölzle
Google

Slower but energy efficient "wimpy" cores only win for general workloads if their single-core speed is reasonably close to that of mid-range "brawny" cores.

At Google, we've been long-term proponents of multicore architectures and throughput-oriented computing. In warehouse-scale systems[1] throughput is more important than single-threaded peak performance, because no single processor can handle the full workload. In addition, maximizing single-threaded performance costs power through larger die areas (for example, for larger reorder buffers or branch predictors) and higher clock frequencies. Multicore architectures are great for warehouse-scale systems because they provide ample parallelism in the request stream as well as data parallelism for search or analysis over petabyte data sets.

We classify multicore systems as *brawny-core systems*, whose single-core performance is fairly high, or *wimpy-core systems*, whose single-core performance is low. The latter are more power efficient. Typically, CPU power decreases by approximately $O(k^2)$ when CPU frequency decreases by k, and decreasing DRAM access speeds with core speeds can save additional power.

So why doesn't everyone want wimpy-core systems? Because in many corners of the real world, they're prohibited by law—Amdahl's law. Even though many Internet services benefit from seemingly unbounded request- and data-level parallelism, such systems aren't above the law. As the number of parallel threads increases, reducing serialization and communication overheads can become increasingly difficult. In a limit case, the amount of inherently serial work performed on behalf of a user request by slow single-threaded cores will dominate overall execution time.

Cost numbers used by wimpy-core evangelists always exclude software development costs. Unfortunately, wimpy-core systems can require applications to be explicitly parallelized or otherwise optimized for acceptable performance. For example, suppose a Web service runs with a latency of one second per user request, half of it caused by serial CPU time. If we switch to wimpy-core servers, whose single-threaded performance is three times slower, the response time doubles to two seconds and developers might have to spend a substantial amount of effort to optimize the code to get back to the one-second latency. Software development costs often dominate a company's overall technical expenses, so forcing programmers to parallelize more code can cost more than we'd save on the hardware side. Most application programmers prefer to think of an individual request as a single-threaded program, leaving the more difficult parallelization problem to middleware that exploits request-level parallelism (that is, it runs independent user requests in separate threads or on separate machines). Once cores become too slow to make this view practical for most applications, you know you're in trouble.

A few other effects splatter more mud on the shiny chrome of wimpy-core designs.

First, the more threads handling a parallelized request, the larger the overall response time. Often all parallel tasks must finish before a request is completed, and thus the overall response time becomes the maximum response time of any subtask, and more subtasks will push further into the long tail of subtask response times. With 10 subtasks, a one-in-a-thousand chance of suboptimal process scheduling will affect 1 percent of requests (recall that the request time is the maximum of all subrequests), but with 1,000 subtasks it will affect virtually all requests.

In addition, a larger number of smaller systems can increase the overall cluster cost if fixed non-CPU costs can't be scaled down accordingly. The cost of basic infrastructure (enclosures, cables, disks, power supplies, network ports, cables, and so on) must be shared across multiple wimpy-core servers, or these costs might offset any savings. More problematically, DRAM costs might increase if processes have a significant DRAM footprint that's unrelated to throughput. For example, the kernel and system processes consume more aggregate memory, and applications can use memory-resident data structures (say, a dictionary mapping words to their synonyms) that might need to be loaded into memory on multiple wimpy-core machines instead of a single brawny-core machine.

Third, smaller servers can also lead to lower utilization. Consider the task of allocating a set of applications across a pool of servers as a bin-packing problem—each of the servers is a bin, and we try to fit as many applications as possible into each bin. Clearly that task is harder when the bins are small, because many applications might not completely fill a server and yet use too much of its CPU or RAM to allow a second application to coexist on the same server. Thus, larger bins (combined with resource containers or virtual machines to achieve performance isolation between individual applications) might offer a lower total cost to run a given workload.

Finally, even embarrassingly parallel algorithms are sometimes intrinsically less efficient when computation and data are partitioned into smaller pieces. That happens, for example, when the stop criterion for a parallel computation is based on global information. To avoid expensive global communication and global lock contention, local tasks can use heuristics that are based on their local progress only, and such heuristics are naturally more conservative. As a result, local subtasks might execute for longer than they would have if better hints about global progress were available. Naturally, when these computations are partitioned into smaller pieces, this overhead tends to increase.

So, although we're enthusiastic users of multicore systems, and believe that throughput-oriented designs generally beat peak-performance-oriented designs, smaller isn't always better. Once a chip's single-core performance lags by more than a factor to two or so behind the higher end of current-generation commodity processors, making a business case for switching to the wimpy system becomes increasingly difficult because application programmers will see it as a significant performance regression: their single-threaded request handlers are no longer fast enough to meet latency targets. So go forth and multiply your cores, but do it in moderation, or the sea of wimpy cores will stick to your programmers' boots like clay.

Reference

1. L. Barroso and U. Hölzle, *The Datacenter as a Computer: An Introduction to Warehouse-Scale Machines*, Morgan Claypool, 2009.

Urs Hölzle *is senior vice president of operations and Google Fellow at Google. His research interests include cluster computing, data centers, networking, and energy efficiency. Hölzle has a PhD in Computer Science from Stanford University. He is a Fellow of the ACM.*

Direct questions and comments about this article to Urs Hölzle, urs@google.com.

To appear in IEEE Micro, 2010.