Web Search for a Planet: The Google Cluster Architecture

AMENABLE TO EXTENSIVE PARALLELIZATION, GOOGLE'S WEB SEARCH APPLICATION LETS DIFFERENT QUERIES RUN ON DIFFERENT PROCESSORS AND, BY PARTITIONING THE OVERALL INDEX, ALSO LETS A SINGLE QUERY USE MULTIPLE PROCESSORS. TO HANDLE THIS WORKLOAD, GOOGLE'S ARCHITECTURE FEATURES CLUSTERS OF MORE THAN 15,000 COMMODITY-CLASS PCS WITH FAULT-TOLERANT SOFTWARE. THIS ARCHITECTURE ACHIEVES SUPERIOR PERFORMANCE AT A FRACTION OF THE COST OF A SYSTEM BUILT FROM FEWER, BUT MORE EXPENSIVE, HIGH-END SERVERS.

Luiz André Barroso
Jeffrey Dean
Urs Hölzle
Google

Few Web services require as much computation per request as search engines. On average, a single query on Google reads hundreds of megabytes of data and consumes tens of billions of CPU cycles. Supporting a peak request stream of thousands of queries per second requires an infrastructure comparable in size to that of the largest supercomputer installations. Combining more than 15,000 commodity-class PCs with fault-tolerant software creates a solution that is more cost-effective than a comparable system built out of a smaller number of high-end servers.

Here we present the architecture of the Google cluster, and discuss the most important factors that influence its design: energy efficiency and price-performance ratio. Energy efficiency is key at our scale of operation, as power consumption and cooling issues become significant operational factors, taxing the limits of available data center power densities.

Our application affords easy parallelization: Different queries can run on different processors, and the overall index is partitioned so that a single query can use multiple processors. Consequently, peak processor performance is less important than its price/performance. As such, Google is an example of a throughput-oriented workload, and should benefit from processor architectures that offer more on-chip parallelism, such as simultaneous multithreading or on-chip multiprocessors.

Google architecture overview

Google's software architecture arises from two basic insights. First, we provide reliability in software rather than in server-class hardware, so we can use commodity PCs to build a high-end computing cluster at a low-end

Published by the IEEE Computer Society

price. Second, we tailor the design for best aggregate request throughput, not peak server response time, since we can manage response times by parallelizing individual requests.

We believe that the best price/performance tradeoff for our applications comes from fashioning a reliable computing infrastructure from clusters of unreliable commodity PCs. We provide reliability in our environment at the software level, by replicating services across many different machines and automatically detecting and handling failures. This software-based reliability encompasses many different areas and involves all parts of our system design. Examining the control flow in handling a query provides insight into the high-level structure of the query-serving system, as well as insight into reliability considerations.

Serving a Google query

When a user enters a query to Google (such as www.google.com/search?q=ieee+society), the user's browser first performs a domain name system (DNS) lookup to map www.google.com to a particular IP address. To provide sufficient capacity to handle query traffic, our service consists of multiple clusters distributed worldwide. Each cluster has around a few thousand machines, and the geographically distributed setup protects us against catastrophic data center failures (like those arising from earthquakes and large-scale power failures). A DNS-based load-balancing system selects a cluster by accounting for the user's geographic proximity to each physical cluster. The load-balancing system minimizes round-trip time for the user's request, while also considering the available capacity at the various clusters.

The user's browser then sends a hypertext transport protocol (HTTP) request to one of these clusters, and thereafter, the processing of that query is entirely local to that cluster. A hardware-based load balancer in each cluster monitors the available set of Google Web servers (GWSs) and performs local load balancing of requests across a set of them. After receiving a query, a GWS machine coordinates the query execution and formats the results into a Hypertext Markup Language (HTML) response to the user's browser. Figure 1 illustrates these steps.

Query execution consists of two major phases.[1] In the first phase, the index servers consult an inverted index that maps each query word to a matching list of documents (the hit list). The index servers then determine a set of relevant documents by intersecting the hit lists of the individual query words, and they compute a relevance score for each document. This relevance score determines the order of results on the output page.

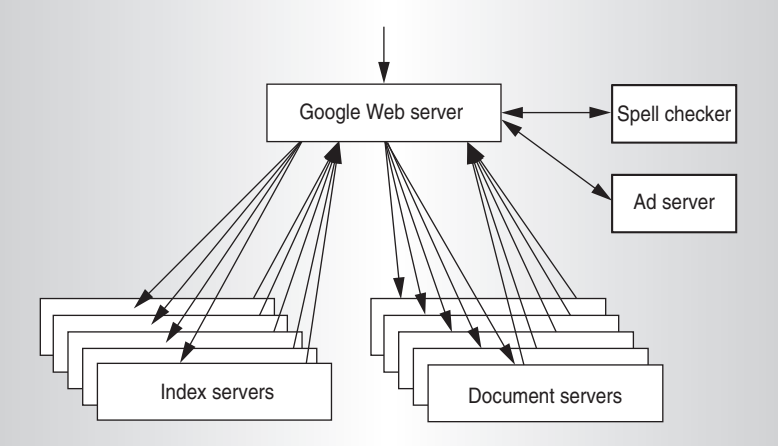

Figure 1. Google query-serving architecture.

The search process is challenging because of the large amount of data: The raw documents comprise several tens of terabytes of uncompressed data, and the inverted index resulting from this raw data is itself many terabytes of data. Fortunately, the search is highly parallelizable by dividing the index into pieces (*index shards*), each having a randomly chosen subset of documents from the full index. A pool of machines serves requests for each shard, and the overall index cluster contains one pool for each shard. Each request chooses a machine within a pool using an intermediate load balancer—in other words, each query goes to one machine (or a subset of machines) assigned to each shard. If a shard's replica goes down, the load balancer will avoid using it for queries, and other components of our cluster-management system will try to revive it or eventually replace it with another machine. During the downtime, the system capacity is reduced in proportion to the total fraction of capacity that this machine represented. However, service remains uninterrupted, and all parts of the index remain available.

The final result of this first phase of query execution is an ordered list of document identifiers (*docids*). As Figure 1 shows, the second

phase involves taking this list of docids and computing the actual title and uniform resource locator of these documents, along with a query-specific document summary. Document servers (docservers) handle this job, fetching each document from disk to extract the title and the keyword-in-context snippet. As with the index lookup phase, the strategy is to partition the processing of all documents by

- randomly distributing documents into smaller shards
- having multiple server replicas responsible for handling each shard, and
- routing requests through a load balancer.

The docserver cluster must have access to an online, low-latency copy of the entire Web. In fact, because of the replication required for performance and availability, Google stores dozens of copies of the Web across its clusters.

In addition to the indexing and document-serving phases, a GWS also initiates several other ancillary tasks upon receiving a query, such as sending the query to a spell-checking system and to an ad-serving system to generate relevant advertisements (if any). When all phases are complete, a GWS generates the appropriate HTML for the output page and returns it to the user's browser.

Using replication for capacity and fault-tolerance

We have structured our system so that most accesses to the index and other data structures involved in answering a query are read-only: Updates are relatively infrequent, and we can often perform them safely by diverting queries away from a service replica during an update. This principle sidesteps many of the consistency issues that typically arise in using a general-purpose database.

We also aggressively exploit the very large amounts of inherent parallelism in the application: For example, we transform the lookup of matching documents in a large index into many lookups for matching documents in a set of smaller indices, followed by a relatively inexpensive merging step. Similarly, we divide the query stream into multiple streams, each handled by a cluster. Adding machines to each pool increases serving capacity, and adding shards accommodates index growth. By parallelizing the search over many machines, we reduce the average latency necessary to answer a query, dividing the total computation across more CPUs and disks. Because individual shards don't need to communicate with each other, the resulting speedup is nearly linear. In other words, the CPU speed of the individual index servers does not directly influence the search's overall performance, because we can increase the number of shards to accommodate slower CPUs, and vice versa. Consequently, our hardware selection process focuses on machines that offer an excellent request throughput for our application, rather than machines that offer the highest single-thread performance.

In summary, Google clusters follow three key design principles:

- *Software reliability.* We eschew fault-tolerant hardware features such as redundant power supplies, a redundant array of inexpensive disks (RAID), and high-quality components, instead focusing on tolerating failures in software.
- *Use replication for better request throughput and availability.* Because machines are inherently unreliable, we replicate each of our internal services across many machines. Because we already replicate services across multiple machines to obtain sufficient capacity, this type of fault tolerance almost comes for free.
- *Price/performance beats peak performance.* We purchase the CPU generation that currently gives the best performance per unit price, not the CPUs that give the best absolute performance.
- *Using commodity PCs reduces the cost of computation.* As a result, we can afford to use more computational resources per query, employ more expensive techniques in our ranking algorithm, or search a larger index of documents.

Leveraging commodity parts

Google's racks consist of 40 to 80 x86-based servers mounted on either side of a custom made rack (each side of the rack contains twenty 20u or forty 1u servers). Our focus on price/performance favors servers that resemble mid-range desktop PCs in terms of their components, except for the choice of large disk

drives. Several CPU generations are in active service, ranging from single-processor 533-MHz Intel-Celeron-based servers to dual 1.4-GHz Intel Pentium III servers. Each server contains one or more integrated drive electronics (IDE) drives, each holding 80 Gbytes. Index servers typically have less disk space than document servers because the former have a more CPU-intensive workload. The servers on each side of a rack interconnect via a 100-Mbps Ethernet switch that has one or two gigabit uplinks to a core gigabit switch that connects all racks together.

Our ultimate selection criterion is cost per query, expressed as the sum of capital expense (with depreciation) and operating costs (hosting, system administration, and repairs) divided by performance. Realistically, a server will not last beyond two or three years, because of its disparity in performance when compared to newer machines. Machines older than three years are so much slower than current-generation machines that it is difficult to achieve proper load distribution and configuration in clusters containing both types. Given the relatively short amortization period, the equipment cost figures prominently in the overall cost equation.

Because Google servers are custom made, we'll use pricing information for comparable PC-based server racks for illustration. For example, in late 2002 a rack of 88 dual-CPU 2-GHz Intel Xeon servers with 2 Gbytes of RAM and an 80-Gbyte hard disk was offered on RackSaver.com for around $278,000. This figure translates into a monthly capital cost of $7,700 per rack over three years. Personnel and hosting costs are the remaining major contributors to overall cost.

The relative importance of equipment cost makes traditional server solutions less appealing for our problem because they increase performance but decrease the price/performance. For example, four-processor motherboards are expensive, and because our application parallelizes very well, such a motherboard doesn't recoup its additional cost with better performance. Similarly, although SCSI disks are faster and more reliable, they typically cost two or three times as much as an equal-capacity IDE drive.

The cost advantages of using inexpensive, PC-based clusters over high-end multiprocessor servers can be quite substantial, at least for a highly parallelizable application like ours. The example $278,000 rack contains 176 2-GHz Xeon CPUs, 176 Gbytes of RAM, and 7 Tbytes of disk space. In comparison, a typical x86-based server contains eight 2-GHz Xeon CPUs, 64 Gbytes of RAM, and 8 Tbytes of disk space; it costs about $758,000.[2] In other words, the multiprocessor server is about three times more expensive but has 22 times fewer CPUs, three times less RAM, and slightly more disk space. Much of the cost difference derives from the much higher interconnect bandwidth and reliability of a high-end server, but again, Google's highly redundant architecture does not rely on either of these attributes.

Operating thousands of mid-range PCs instead of a few high-end multiprocessor servers incurs significant system administration and repair costs. However, for a relatively homogenous application like Google, where most servers run one of very few applications, these costs are manageable. Assuming tools to install and upgrade software on groups of machines are available, the time and cost to maintain 1,000 servers isn't much more than the cost of maintaining 100 servers because all machines have identical configurations. Similarly, the cost of monitoring a cluster using a scalable application-monitoring system does not increase greatly with cluster size. Furthermore, we can keep repair costs reasonably low by batching repairs and ensuring that we can easily swap out components with the highest failure rates, such as disks and power supplies.

The power problem

Even without special, high-density packaging, power consumption and cooling issues can become challenging. A mid-range server with dual 1.4-GHz Pentium III processors draws about 90 W of DC power under load: roughly 55 W for the two CPUs, 10 W for a disk drive, and 25 W to power DRAM and the motherboard. With a typical efficiency of about 75 percent for an ATX power supply, this translates into 120 W of AC power per server, or roughly 10 kW per rack. A rack comfortably fits in 25 ft^2 of space, resulting in a power density of 400 W/ft^2. With higher-end processors, the power density of a rack can exceed 700 W/ft^2.

Table 1. Instruction-level measurements on the index server.

Characteristic	Value
Cycles per instruction	1.1
Ratios (percentage)	
Branch mispredict	5.0
Level 1 instruction miss*	0.4
Level 1 data miss*	0.7
Level 2 miss*	0.3
Instruction TLB miss*	0.04
Data TLB miss*	0.7

* Cache and TLB ratios are per instructions retired.

Unfortunately, the typical power density for commercial data centers lies between 70 and 150 W/ft^2, much lower than that required for PC clusters. As a result, even low-tech PC clusters using relatively straightforward packaging need special cooling or additional space to bring down power density to that which is tolerable in typical data centers. Thus, packing even more servers into a rack could be of limited practical use for large-scale deployment as long as such racks reside in standard data centers. This situation leads to the question of whether it is possible to reduce the power usage per server.

Reduced-power servers are attractive for large-scale clusters, but you must keep some caveats in mind. First, reduced power is desirable, but, for our application, it must come without a corresponding performance penalty: What counts is watts per unit of performance, not watts alone. Second, the lower-power server must not be considerably more expensive, because the cost of depreciation typically outweighs the cost of power. The earlier-mentioned 10 kW rack consumes about 10 MW-h of power per month (including cooling overhead). Even at a generous 15 cents per kilowatt-hour (half for the actual power, half to amortize uninterruptible power supply [UPS] and power distribution equipment), power and cooling cost only $1,500 per month. Such a cost is small in comparison to the depreciation cost of $7,700 per month. Thus, low-power servers must not be more expensive than regular servers to have an overall cost advantage in our setup.

Hardware-level application characteristics

Examining various architectural characteristics of our application helps illustrate which hardware platforms will provide the best price/performance for our query-serving system. We'll concentrate on the characteristics of the index server, the component of our infrastructure whose price/performance most heavily impacts overall price/performance. The main activity in the index server consists of decoding compressed information in the inverted index and finding matches against a set of documents that could satisfy a query. Table 1 shows some basic instruction-level measurements of the index server program running on a 1-GHz dual-processor Pentium III system.

The application has a moderately high CPI, considering that the Pentium III is capable of issuing three instructions per cycle. We expect such behavior, considering that the application traverses dynamic data structures and that control flow is data dependent, creating a significant number of difficult-to-predict branches. In fact, the same workload running on the newer Pentium 4 processor exhibits nearly twice the CPI and approximately the same branch prediction performance, even though the Pentium 4 can issue more instructions concurrently and has superior branch prediction logic. In essence, there isn't that much exploitable instruction-level parallelism (ILP) in the workload. Our measurements suggest that the level of aggressive out-of-order, speculative execution present in modern processors is already beyond the point of diminishing performance returns for such programs.

A more profitable way to exploit parallelism for applications such as the index server is to leverage the trivially parallelizable computation. Processing each query shares mostly read-only data with the rest of the system, and constitutes a work unit that requires little communication. We already take advantage of that at the cluster level by deploying large numbers of inexpensive nodes, rather than fewer high-end ones. Exploiting such abundant thread-level parallelism at the microarchitecture level appears equally promising. Both simultaneous multithreading (SMT) and chip multiprocessor (CMP) architectures target thread-level parallelism and should improve the performance of many of our servers. Some early

experiments with a dual-context (SMT) Intel Xeon processor show more than a 30 percent performance improvement over a single-context setup. This speedup is at the upper bound of improvements reported by Intel for their SMT implementation.[3]

We believe that the potential for CMP systems is even greater. CMP designs, such as Hydra[4] and Piranha,[5] seem especially promising. In these designs, multiple (four to eight) simpler, in-order, short-pipeline cores replace a complex high-performance core. The penalties of in-order execution should be minor given how little ILP our application yields, and shorter pipelines would reduce or eliminate branch mispredict penalties. The available thread-level parallelism should allow near-linear speedup with the number of cores, and a shared L2 cache of reasonable size would speed up interprocessor communication.

Memory system

Table 1 also outlines the main memory system performance parameters. We observe good performance for the instruction cache and instruction translation look-aside buffer, a result of the relatively small inner-loop code size. Index data blocks have no temporal locality, due to the sheer size of the index data and the unpredictability in access patterns for the index's data block. However, accesses within an index data block do benefit from spatial locality, which hardware prefetching (or possibly larger cache lines) can exploit. The net effect is good overall cache hit ratios, even for relatively modest cache sizes.

Memory bandwidth does not appear to be a bottleneck. We estimate the memory bus utilization of a Pentium-class processor system to be well under 20 percent. This is mainly due to the amount of computation required (on average) for every cache line of index data brought into the processor caches, and to the data-dependent nature of the data fetch stream. In many ways, the index server's memory system behavior resembles the behavior reported for the Transaction Processing Performance Council's benchmark D (TPC-D).[6] For such workloads, a memory system with a relatively modest sized L2 cache, short L2 cache and memory latencies, and longer (perhaps 128 byte) cache lines is likely to be the most effective.

Large-scale multiprocessing

As mentioned earlier, our infrastructure consists of a massively large cluster of inexpensive desktop-class machines, as opposed to a smaller number of large-scale shared-memory machines. Large shared-memory machines are most useful when the computation-to-communication ratio is low; communication patterns or data partitioning are dynamic or hard to predict; or when total cost of ownership dwarfs hardware costs (due to management overhead and software licensing prices). In those situations they justify their high price tags.

At Google, none of these requirements apply, because we partition index data and computation to minimize communication and evenly balance the load across servers. We also produce all our software in-house, and minimize system management overhead through extensive automation and monitoring, which makes hardware costs a significant fraction of the total system operating expenses. Moreover, large-scale shared-memory machines still do not handle individual hardware component or software failures gracefully, with most fault types causing a full system crash. By deploying many small multiprocessors, we contain the effect of faults to smaller pieces of the system. Overall, a cluster solution fits the performance and availability requirements of our service at significantly lower costs.

At first sight, it might appear that there are few applications that share Google's characteristics, because there are few services that require many thousands of servers and petabytes of storage. However, many applications share the essential traits that allow for a PC-based cluster architecture. As long as an application orientation focuses on the price/performance and can run on servers that have no private state (so servers can be replicated), it might benefit from using a similar architecture. Common examples include high-volume Web servers or application servers that are computationally intensive but essentially stateless. All of these applications have plenty of request-level parallelism, a characteristic exploitable by running individual requests on separate servers. In fact, larger Web sites already commonly use such architectures.

At Google's scale, some limits of massive server parallelism do become apparent, such as the limited cooling capacity of commercial data centers and the less-than-optimal fit of current CPUs for throughput-oriented applications. Nevertheless, using inexpensive PCs to handle Google's large-scale computations has drastically increased the amount of computation we can afford to spend per query, thus helping to improve the Internet search experience of tens of millions of users. MICRO

Acknowledgments

Over the years, many others have made contributions to Google's hardware architecture that are at least as significant as ours. In particular, we acknowledge the work of Gerald Aigner, Ross Biro, Bogdan Cocosel, and Larry Page.

References

1. S. Brin and L. Page, "The Anatomy of a Large-Scale Hypertextual Web Search Engine," *Proc. Seventh World Wide Web Conf.* (WWW7), International World Wide Web Conference Committee (IW3C2), 1998, pp. 107-117.
2. "TPC Benchmark C Full Disclosure Report for IBM eserver xSeries 440 using Microsoft SQL Server 2000 Enterprise Edition and Microsoft Windows .NET Datacenter Server 2003, TPC-C Version 5.0," http://www.tpc.org/results/FDR/TPCC/ibm.x4408way.c5.fdr.02110801.pdf.
3. D. Marr et al., "Hyper-Threading Technology Architecture and Microarchitecture: A Hypertext History," *Intel Technology J.*, vol. 6, issue 1, Feb. 2002.
4. L. Hammond, B. Nayfeh, and K. Olukotun, "A Single-Chip Multiprocessor," *Computer*, vol. 30, no. 9, Sept. 1997, pp. 79-85.
5. L.A. Barroso et al., "Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing," *Proc. 27th ACM Int'l Symp. Computer Architecture*, ACM Press, 2000, pp. 282-293.
6. L.A. Barroso, K. Gharachorloo, and E. Bugnion, "Memory System Characterization of Commercial Workloads," *Proc. 25th ACM Int'l Symp. Computer Architecture*, ACM Press, 1998, pp. 3-14.

Luiz André Barroso is a member of the Systems Lab at Google, where he has focused on improving the efficiency of Google's Web search and on Google's hardware architecture. Barroso has a BS and an MS in electrical engineering from Pontifícia Universidade Católica, Brazil, and a PhD in computer engineering from the University of Southern California. He is a member of the ACM.

Jeffrey Dean is a distinguished engineer in the Systems Lab at Google and has worked on the crawling, indexing, and query serving systems, with a focus on scalability and improving relevance. Dean has a BS in computer science and economics from the University of Minnesota and a PhD in computer science from the University of Washington. He is a member of the ACM.

Urs Hölzle is a Google Fellow and in his previous role as vice president of engineering was responsible for managing the development and operation of the Google search engine during its first two years. Hölzle has a diploma from the Eidgenössische Technische Hochschule Zürich and a PhD from Stanford University, both in computer science. He is a member of IEEE and the ACM.

Direct questions and comments about this article to Urs Hölzle, 2400 Bayshore Parkway, Mountain View, CA 94043; urs@google.com.