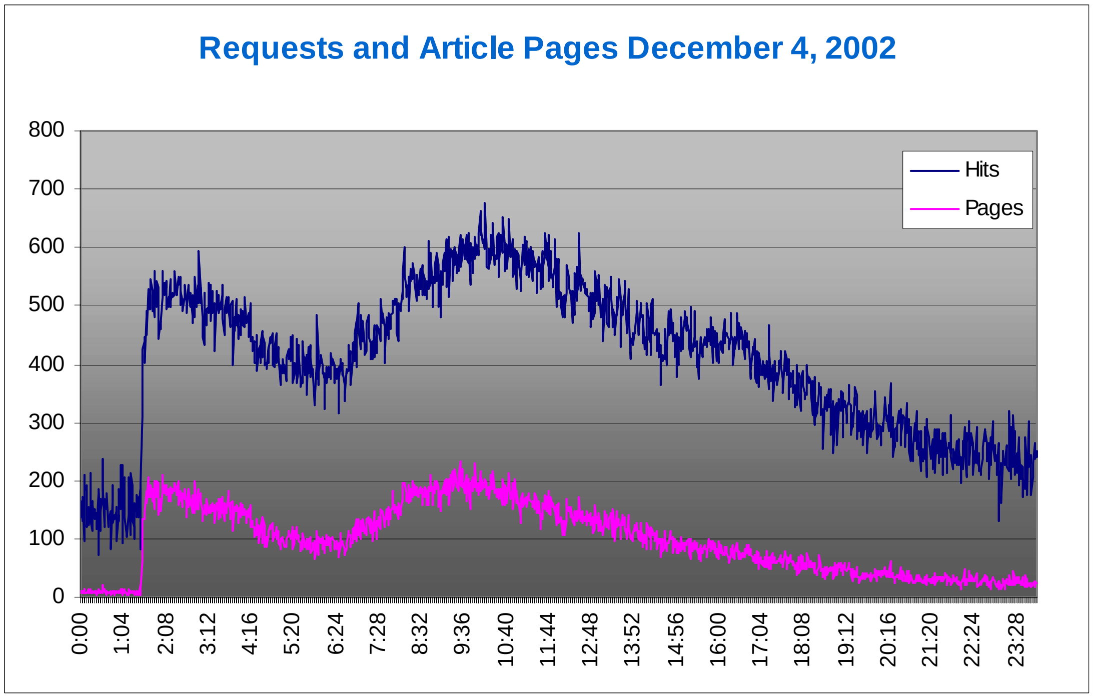
Scaling Server Performance

By Brian Neal – January 2003

Where Performance is Concerned, Optimization is Key

When our Hitchhiker's Guide to the Mainframe article caught the attention of Slashdot last month, quite a few people were curious to know about how our server handled the traffic. This is a topic we have discussed previously in Building a Better Webserver in the 21st Century and SPECmine - A Case Study on Optimization, but since there was so much interest in some more in-depth information on the topic, I decided to spend some time explaining how the data object caching in our web application can do so much for performance without sacrificing the ability to serve true dynamic content. I'll start with some statistics gathered on December 4th.

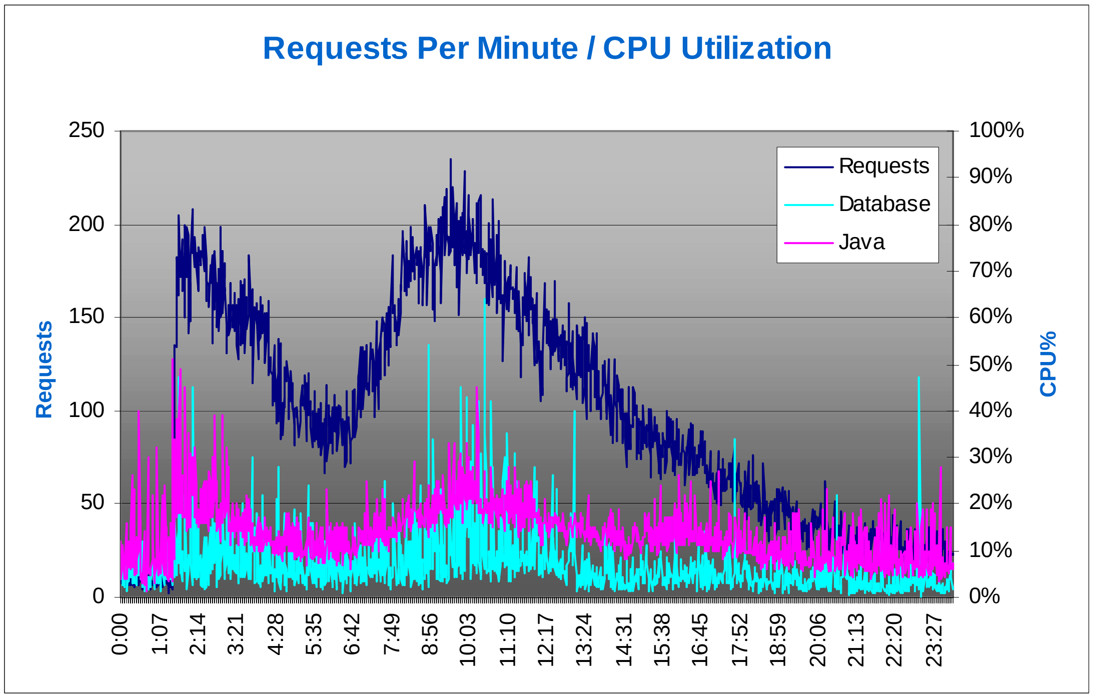
Our traffic for the day totaled over 590,000 requests (hits), with over 250,000 of those being requests for pages. Requests peaked at 677 per minute, which of course includes everything (images, pages, files, etc.). The peak number strictly for article pages was 235 per minute. Perhaps the most impressive statistic is that during these peaks, our web application running in Java (including the HTTP server) was only consuming 20% of available CPU time, and all article requests were served in 4 milliseconds or less. Furthermore, our database was only at 7.5% CPU utilization. So, when the system was serving roughly 11 requests per second, the CPU was nearly 75% idle. Let's take a look at the traffic for the day, graphed on a per-minute basis:



In the graph above, we have two different sets of data. The first is requests, which is essentially anything requested from the server -- images, dynamic pages, static pages, etc. The second is article pages, like this one.

Copyright © 1998-2003 Ace's Hardware. All Rights Reserved.

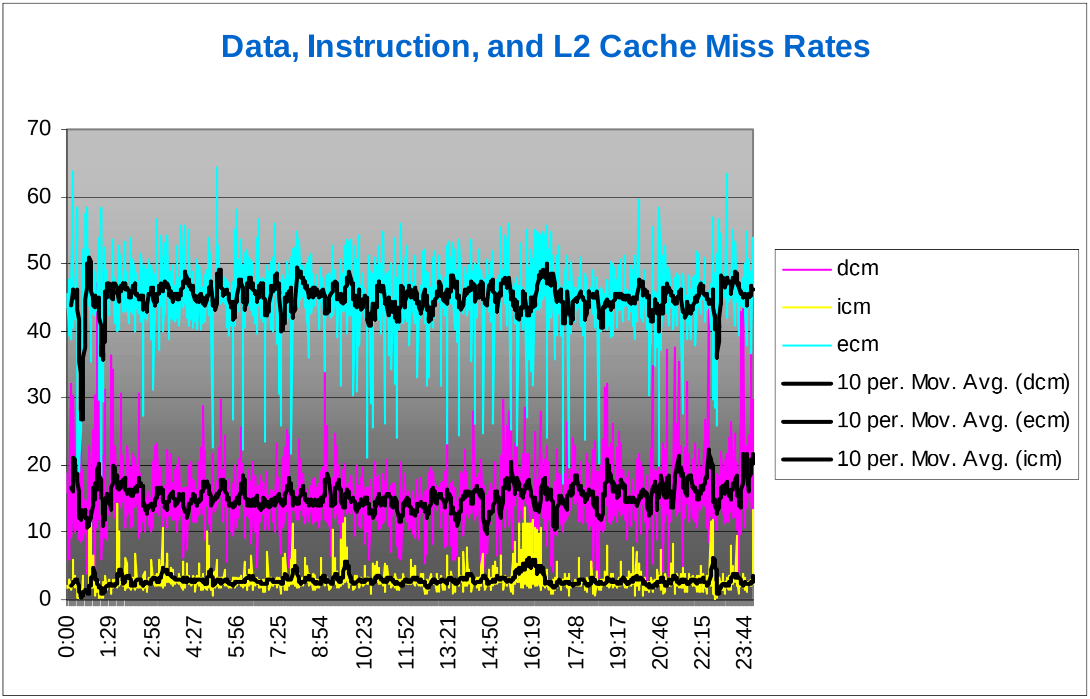
As you can see, the initial spike in traffic from Slashdot occurs around 1:30 AM. There is other dynamic content, such as the front page or the message board, that is not accounted for in this data, but nevertheless, the graph gives you a good idea of the ratio between article pages (text) and everything else. Traditionally, dynamic content is often one of the most intensive types of content for a webserver, but as you will see in the next graph, that doesn't always have to be the case.



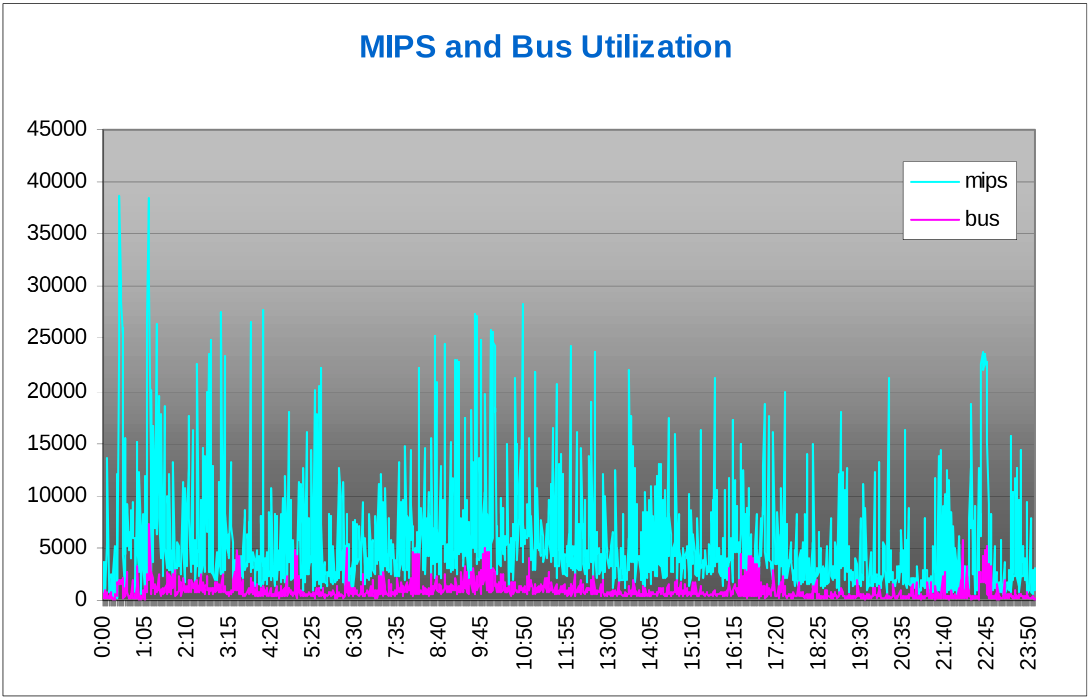
In this graph we see the CPU utilization of both the Java web application and the database, sampled each minute, relative to the article requests per minute. Here you can clearly see the peaks I mentioned earlier happening roughly between 9:30 and 10:00 AM. You can also see that the average CPU utilization of both the web application and the database is rather low: 14% for Java and 6.9% for the database. The Java CPU utilization peaks at 51% for a very short time early on, as it is caching content on demand as traffic begins to spike. You can also see occasional spikes in the database CPU utilization, as cached data is periodically updated. When you consider that there are servers out there that will fall over and die when faced with this kind of traffic, an overall average CPU utilization of 21% for a modest 550 MHz uniprocessor machine is not too shabby.

We monitor and collect statistics for quite a few different components of our server's performance. Thanks to the presence of hardware performance counters in Solaris, we can monitor a number of CPU-level statistics, including average cycles per instruction, bus utilization, branch-misprediction rate, cache miss rate, etc. As with the other performance data we record, these statistics are collected in 1-minute intervals.

Copyright © 1998-2003 Ace's Hardware. All Rights Reserved.



Above is the data, instruction, and L2 cache miss rates of the server CPU, graphed over the course of the day for December 4, 2002 in 1 minute intervals. As this is a rather crowded graph, trendlines have been added to improve visibility. Aside from some occasional dips and spikes, the miss rates remain rather consistent throughout the day. Keep in mind that although the traffic was considerable, the overall CPU utilization remained rather low.



Here we see the processor MIPS and address bus utilization. Once again, given that the CPU load averages only around 20% and remains fairly consistent, there aren't any tremendously notable variations in most of the data from the performance counters.

Copyright © 1998-2003 Ace's Hardware. All Rights Reserved.

Scaling with Larger Workloads Effectively

As you may have guessed by now, the key to our server performance does not lie in the physical hardware itself, but in the software. Specifically, our web application is designed from the ground up to serve even some of our most complex dynamic pages in milliseconds. To accomplish this, you have to isolate the bottlenecks in your software and eliminate them. Running complex SQL queries on every request, for instance, can be a severe limit to a server's capacity. Even simple queries can be a problem if the demand is high enough.

To take this example a little further, it's important to realize that the problem is not just the query, but all the steps required to even be able to run the query. This includes the time required to allocate a database connection, or create one if none are available from a connection pool, the time required to optimize and compile the query (unless it's a stored procedure), and the time required to generate the result and send it back to the application.

Aside from the query time, this added complexity also introduces some potential bottlenecks depending upon the situation. Since it is necessary to allocate a connection to communicate with the database, we have to consider how such connections are allocated in order to avoid possible limits on the number of concurrent connections and make sure the total number of open connections does not consume too much memory. Aside from this, there are also other query-dependent issues, such as table locks. If the application stalls under high load as a result of an exclusive table lock or a database connection limit, there may be serious consequences for performance.

Software Bottlenecks

In the webserver article I mentioned earlier, I described a situation where a server running a database-driven dynamic site on Apache 1.3.x and PHP could literally DoS itself when subjected to high loads (like those from Slashdot). In that situation, the database connections are poorly distributed amongst individual HTTP server processes. To improve performance, these connections can be made to persist between requests, but since HTTP is a stateless protocol and each individual process is unware of the others, the persistent database connection is permanently associated with that specific HTTP process and can only be used for requests handled by it.

This differs from traditional connection pooling where all open connections are pooled and shared between all threads/processes. In the case of a connection pool, a connection can be allocated from the pool when needed and then later returned so that it may be used by other processes.

Back to the Apache 1.3.x example, we have a large number of HTTP processes with open database connections that are very likely, at any given moment, not serving requests requiring interaction with the database. Consider the graph shown earlier, comparing requests for article pages with all other requests. If the server is serving a dynamic page with five images in it, then there will be five static requests for each dynamic one.

There's another specific issue with Apache 1.3.x regarding how keepalives are handled. This is a feature designed to improve performance by holding the connection to the client open following a request for a short time in the event that further requests are made. This makes sense, as individual pages often need multiple requests to be downloaded in the case of images and so forth. However, this also means that each HTTP process must wait idle for a certain period of time, perhaps 10 or 15 seconds, before accepting requests from a different client. When a server encounters a large number of concurrent requests from unique clients, this behavior can result in a large number of idle HTTP processes waiting to timeout.

The effect is cumulative. A machine serving dynamic database-driven pages is under high load, so it spawns more processes to handle all the concurrent requests. Since the persistent database connections in use by existing HTTP processes are not universally available, the newly created processes have to open more database connections, even if others are not currently in use. Additionally, since keepalives are enabled and there are a large number of unique clients, a significant portion of the HTTP process pool is idle and cannot serve new clients until they timeout. This means even more HTTP processes have to be created and even more database connections have to be opened. The real question in this situation is what resource will the server run out of first: connections or memory?

Copyright © 1998-2003 Ace's Hardware. All Rights Reserved.

Faster or More Efficient?

It's true that you can fight bottlenecks like these with hardware, but it's an expensive proposition. In the case described above, you could need two or three times the hardware, depending upon the amount of traffic, or even more. If we assume for the sake of argument that the application load is distributed evenly amongst all the server's major components, then this means you would need to increase CPU performance, memory size (and performance), and disk performance proportionally. This could be accomplished by upgrading the CPUs, memory, and disks/storage array or by clustering. And all this is just to handle the existing load.

Configuring the Ace's Hardware Web Server...

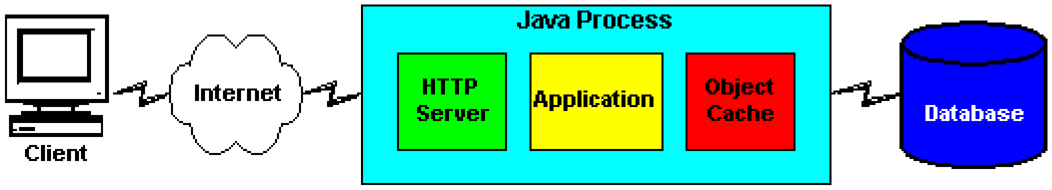
We made some substantial upgrades to our single-thread compute performance and memory when we replaced our old server last year. Specifically, we went from four 125 MHz CPUs with 512 MB of RAM to a single 550 MHz CPU with 2 GB of RAM. However, the increased server performance did not reduce the average article load time from 4 seconds to 4 milliseconds. To accomplish this, our web application had to be reimplemented from the ground up.

As mentioned earlier, you can mitigate performance issues by throwing hardware at them. If database performance is lacking, a fast RAID array with a large cache can alleviate the problem. But even a system with solid state disks (SSDs) can't outperform a system that doesn't have to run the query in the first place.

Copyright © 1998-2003 Ace’s Hardware. All Rights Reserved.

Inside the Web Server Application

By caching data on demand inside the application, we can avoid the database altogether. This means no need to allocate a database connection, no time spent compiling/optimizing the query, no time spent running the query and returning the results, and no inter-process communication (IPC). By eliminating the bottleneck, we can avoid all the potential consequences that bottleneck presents in high-load situations. Let's take a look at what happens inside the application when a client requests an article:



Above is a diagram describing the relationship between the client, server application, and database. As you can see, the HTTP server, application logic, and data object cache are all in the same process running on the server. The data object caching is possible because the application is persistent, rather than just a collection of scripts executed in a stateless manner, and can share objects between any and all requests. Communication between these components is very quick relative to talking to the database or (much more so) the client. Now, keeping this diagram in mind, let's break the request down step-by-step:

Step 1.	GET /read.jsp?id=50000333
Step 2.	ArticleClass articleData = cache.getId(50000333);
Step 3.	if(articleData == NULL) {

Step 3a.	JDBConnection con = pool.get();
	ArticleClass articleData = con.query("SELECT * from articles");

Step 3b.	cache.add(articleData);

Step 4.	}
	generatePage(articleData);

Step 5.	HTTP/1.0 200 OK Content-Type: text/html

Each general step in the process has been color-coded according to the diagram above and includes pseudo-code describing what is actually occurring. The request is initiated by the client, which talks to the HTTP server. The application checks to see whether or not the requested article is cached. If so, the page is generated from the cached data and then sent to the client.

If the requested article is not cached, then a database connection is allocated from the connection pool and a query is executed to retrieve the article from the database. The data from the query is stored in an object for use by the application and this object is subsequently added to the cache for future requests.

Copyright © 1998-2003 Ace's Hardware. All Rights Reserved.

Benchmarking with ApacheBench

So, just how much time do we save by avoiding the database access? To find out, I benchmarked five article requests. Each article was requested twice. On the first request, the article was uncached, meaning the time required to allocate a database connection, run a query, and get the result was benchmarked along with the rest of the process. On the second request, the article was cached, so only the time required to retrieve the article from cache and generate the page was benchmarked. The results shown below were obtained from our server running with minimal load (> 95% idle):

Speed of Cached VS Uncached Articles

Test	Database Time	Total Time (Uncached)	Total Time (Cached)	Difference
1	134 ms	138 ms	3 ms	4500%
2	118 ms	121 ms	4 ms	2925%
3	167 ms	171 ms	4 ms	4175%
4	214 ms	217 ms	4 ms	5325%
5	161 ms	165 ms	3 ms	5400%
Average	**159 ms**	**162 ms**	**3.6 ms**	**4400%**

As you can see, the difference between the overall time for the cached and uncached pages is tremendous. Of course, one might say that even in the worst case, the uncached time is only two tenths of a second. However, two decimal places is a big difference and it really adds up under high loads. The cached version will scale a lot better as well. Let's see the results in the case of a loaded server:

1000 Requests, 18.8 KB Article

ApacheBench Results					Connection Times		
Benchmark	**Total Req.**	**Concurrent Req.**	**Time**	**Requests/Sec.**	**min**	**avg**	**max**
Uncached	1000	1	130.419 s	7.67	116 ms	129 ms	772 ms
Cached	1000	1	16.799 s	59.53	13 ms	15 ms	159 ms
Uncached	1000	10	33.242 s	30.08	132 ms	329 ms	877 ms
Cached	1000	10	15.522 s	64.42	13 ms	149 ms	979 ms

Here, we're using ApacheBench to benchmark the server using the first page of "A Quick Look at the Fastest Apple PowerMac." Both cached and uncached versions of the article page are tested, with no concurrency as well as results from benchmarking with 10 concurrent connections. The connection times are ApacheBench's total connection times. The benchmark was performed locally on the server during an unloaded period and each result is the best of three runs.

As you can see, the cached version is many times faster than the uncached version, though that lead is diminished when testing with 10 concurrent connections. With no concurrency, the uncached version handles a little less than 8 connections per second. The cached version, on the other hand, serves almost 60 connections per second. With 10 concurrent connections, the uncached version manages 30 requests per second, while the cached version is a little better than twice as fast at 64 connections per second.

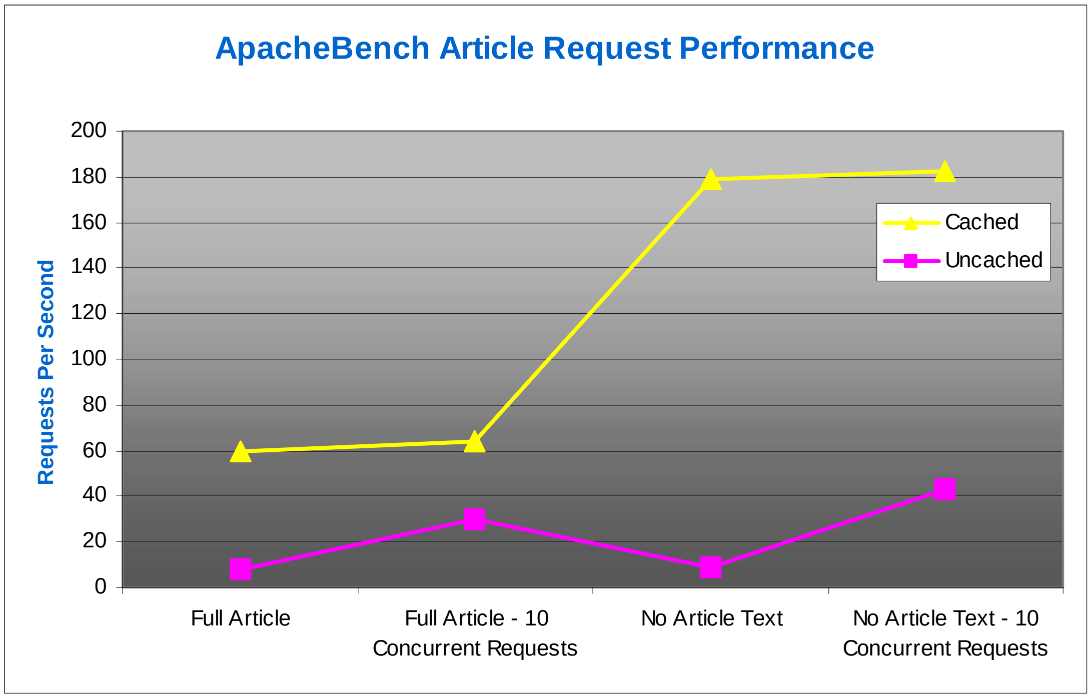
Unfortunately, ApacheBench does not support compressed content, meaning our test was benchmarking pages significantly larger than the average article page our server usually transfers, as most modern browsers support compression. Therefore, I ran another set of benchmarks, this time using pages that did not display the actual article text. This reduced the request size from 18 KB to less than 1 KB (the actual compressed article page is roughly 5 KB). Even though the article text is not included in the page, it is still retrieved (from cache or database) and parsed/formatted as usual. So, this benchmark still presents the same computational load to the server as before, but alleviates bandwidth as a possible bottleneck and minimizes the HTTP workload.

Copyright © 1998-2003 Ace's Hardware. All Rights Reserved.

1000 Requests, 933 byte Article (No Article Text)

ApacheBench Results					**Connection Times**		
Benchmark	**Total Req.**	**Concurrent Req.**	**Time**	**Requests/Sec.**	**min**	**avg**	**max**
Uncached	1000	1	118.671 s	8.43	106 ms	117 ms	641 ms
Cached	1000	1	5.577 s	179.31	4 ms	4 ms	64 ms
Uncached	1000	10	23.387 s	42.76	111 ms	231 ms	1166 ms
Cached	1000	10	5.464 s	182.88	8 ms	53 ms	509 ms

Both versions see improvements, but the performance of the cached version has tripled. It is now handling roughly 180 requests per second while the uncached version serves a little over 40 requests per second. For a less cluttered view of the comparison, take a look at the following graph:

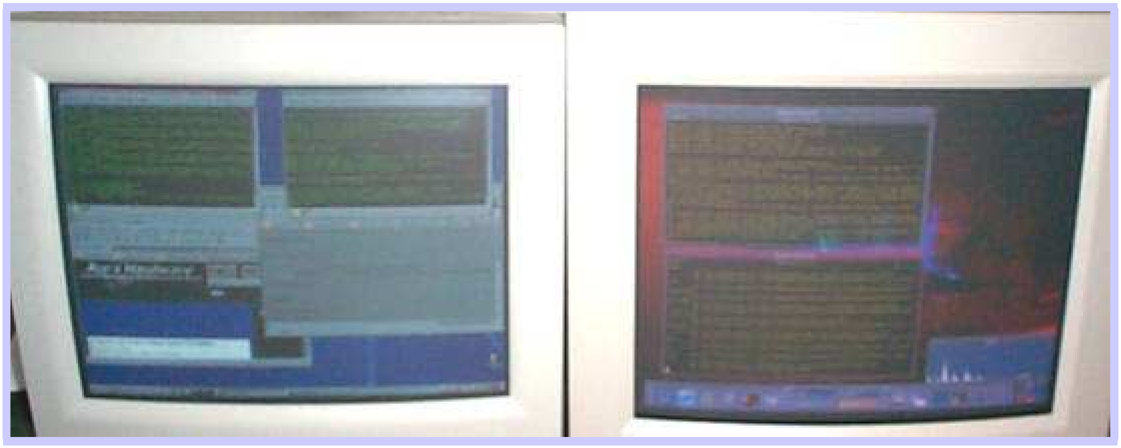


The ApacheBench results have given us a good idea as to the performance differences between plain database driven article requests and article requests augmented by a data object cache. But what about scalability, and how does our chosen HTTP server and servlet engine, Resin (a high-performance Java application server with downloadable source), stack up to Apache? To find out, I configured dedicated client and server systems on a 100Mbit switched network for testing. The server was configured with the same software we run in our production environment, and the database was populated with all the same data.

Copyright © 1998-2003 Ace's Hardware. All Rights Reserved.

Benchmarking Web Server Scalability

The server hardware for the following tests is a 300 MHz Ultra 30 (2 MB L2 cache) running Solaris 9, configured with 386 MB of memory and a 7200 RPM Seagate Barracuda hard disk. This system has a slower CPU and less memory compared to our production server, but this just means it will be saturated sooner. The client system is a 2.8 GHz Pentium 4 on an ASUS P4T533, configured with 512 MB of RAM, running Mandrake Linux 9.



Client on the left, server on the right...

For benchmarking, httperf was used in conjunction with autobench, a Perl script designed to run httperf against a server several times, with the number of requests per second increasing with each iteration. The output from the program enables us to see exactly how well the system being tested performs as the workload is gradually increased until it becomes saturated.

In each case, the server was benchmarked at an initial rate of 10 connections per second with 5 requests per connection. This initial rate of 50 requests per second was increased by 50 for each iteration until reaching a total of 1500 requests per second. The benchmark was configured to timeout any request that takes longer than 5 seconds to complete.

The benchmarked HTTP servers included:

- Caucho Technology's Resin 2.0.2
- Apache 1.3.27
- Apache 2.0.43

The most recent versions of all servers at the time of writing were tested with the exception of Resin. In this instance, we benchmarked the exact same configuration we use in production. Theoretically, this puts Resin at something of a disadvantage considering both Apache 1.3.27 and Apache 2.0.43 were compiled specifically for benchmarking with a minimum of modules you might otherwise expect to find on a production Apache server (server side includes, user directories, automatic directory indexes, logging, etc.). The final size of the stripped Apache 1.3.27 binary was 589 KB. The Apache 2.0.43 binary was 622 KB and was compiled with the worker MPM. Resin was benchmarked with the native/in-process HTTP server and ran under the Java HotSpot 1.4.0 Server VM.

Copyright © 1998-2003 Ace's Hardware. All Rights Reserved.

Key configuration parameters for all the servers follow:

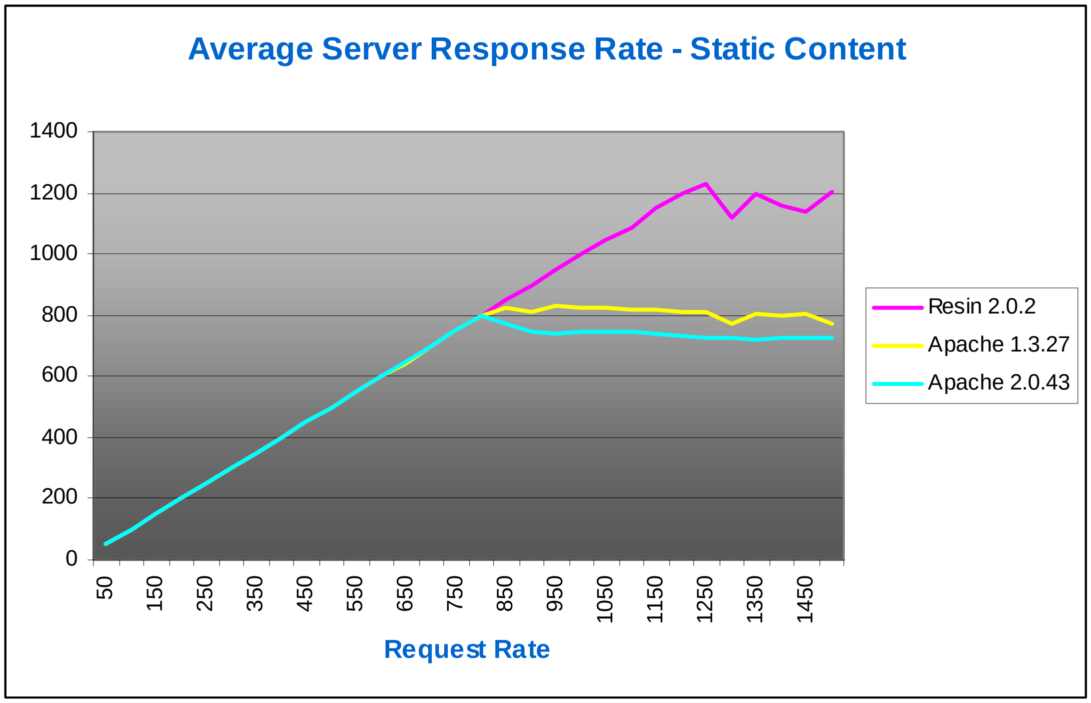
- **Resin 2.0.2**
  - thread-min 10
  - thread-max 500
  - ignore-client-disconnect false
- **Apache 1.3.27**
  - MinSpareServers 5
  - MaxSpareServers 10
  - StartServers 5
  - MaxClients 700
  - MaxRequestsPerChild 0
  - KeepAlive On
  - MaxKeepAliveRequests 500
  - KeepAliveTimeout 15
- **Apache 2.0.43**
  - StartServers 2
  - MaxClients 150
  - MinSpareThreads 25
  - MaxSpareThreads 75
  - ThreadsPerChild 25
  - MaxRequestsPerChild 0
  - KeepAlive On
  - MaxKeepAliveRequests 500
  - KeepAliveTimeout 15

Before benchmarking, each server was tested with httperf at 1500 requests per second to make sure that the server would be saturated by this rate and to allow the server to build up any runtime caches, thread/process pools, etc. The maximum number of file descriptors was set to 8192 on both the client and server machines.

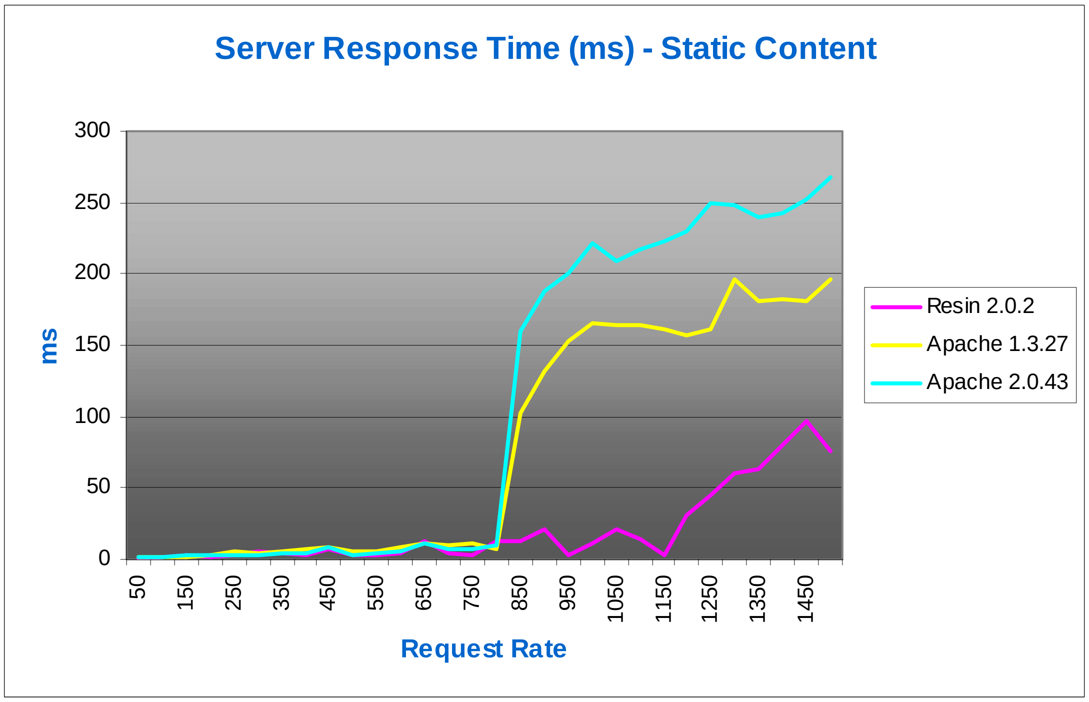
Static HTTP Performance

Serving static content is the least demanding workload a modern day HTTP server will encounter. There's no dynamic content to create, text to format, or databases to query -- just files to transfer. Even though many more complex sites on the Internet today serve dynamic content, static content is still very common in the form of images, file downloads, and so forth. For our first test, we'll compare the static HTTP performance of all three servers using a 67 byte GIF file. The servers are benchmarked from 50 to 1500 requests per second.

Copyright © 1998-2003 Ace's Hardware. All Rights Reserved.



As you can see, the request rate begins at 50 requests per second and increases steadily until it reaches 1500 requests per second. All three servers match the demanded workload up until roughly 800 connections per second. At this point, the Apache servers become saturated. Apache 1.3.27 sustains a maximum of 828 requests per second, while Apache 2.0.43 actually becomes saturated slightly lower at a maximum of 799 requests per second. Resin 2.0.2 continues on and maxes out at 1232 requests per second. Let's compare the response times:



Here we looking at the average amount of time each server requires before sending a response, so lower is better. Again, the response time remains similar for all three servers until the request rate reaches roughly 800 requests per second.

Copyright © 1998-2003 Ace's Hardware. All Rights Reserved.

Beyond that point, the response time for the Apache servers begins to increase as they become saturated. At 850 requests per second, the average response time is 102 ms for Apache 1.3.27. At 1200 requests per second it is 157 ms, and at 1500 requests per second it is 196 ms.

The results are similar for Apache 2.0.43, though the response times are higher. At 850 requests per second, the response time is 159 ms. This grows to 230 ms at 1200 requests per second and ultimately to 267 ms at 1500 requests per second.

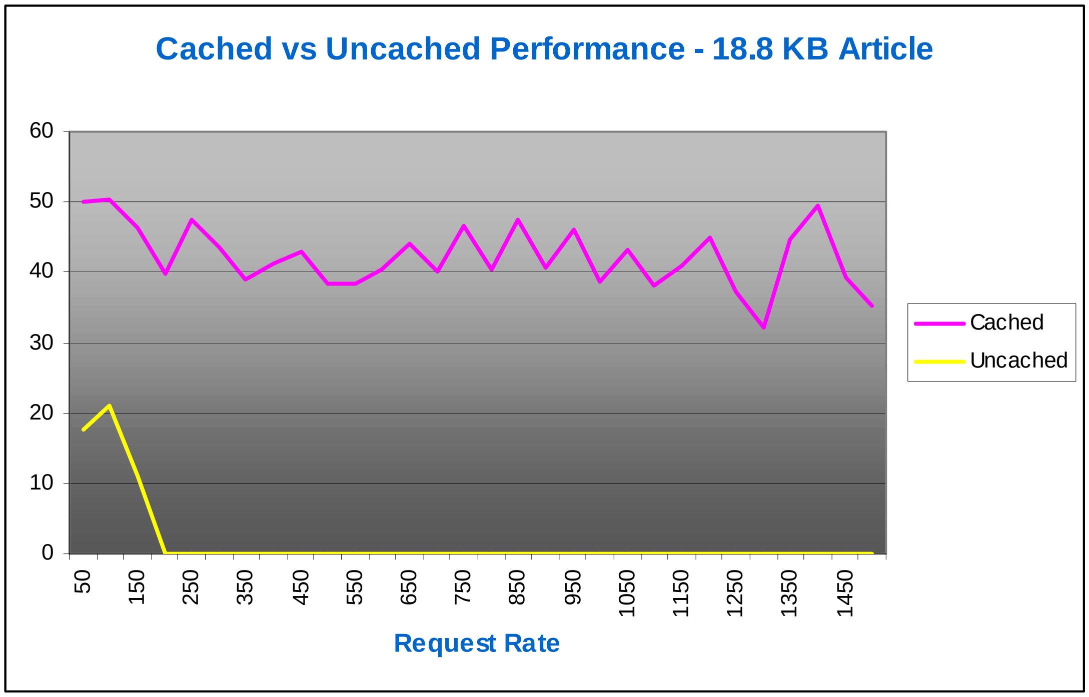
The Resin response time fluctuates somewhat at higher request rates, but manages to stay well below that of the other servers. At 850 requests per second, the response time for Resin is 12.7 ms. At 1200 requests per second, the response time roughly doubles to 30 ms before finally reaching a maximum of 96 ms at 1450 requests per second. This is half the maximum request time of Apache 1.3.27 and nearly a third of Apache 2.0.43's highest time.

It's interesting that despite the move to a worker thread model, Apache 2.0's static HTTP performance has not improved over that of Apache 1.3. The server is, however, still relatively new. Julian T. J. Midgley, the author of autobench, discusses Apache 2.0's performance relative to Apache 1.3:

> "As someone else has said, Apache 2.0 doesn't give much of a speed advantage over Apache 1, and certainly doesn't approach Zeus's performance, since it variously relies on a multi-threaded or multi-process approach to connection handling. Even the multi-threaded MPM on a OS with decent thread support, such as Solaris, falls foul of the context-switching problem before the number of connections has risen much above those that Apache 1.0 can cope with."

Dynamic Cached vs Uncached Article Performance

Apache 2.0 may have room for improvement, but for the moment, Resin has managed to outperform both it and the older Apache 1.3 by a respectable margin in static HTTP serving. This article began with a discussion on the performance of dynamic content, however, and the advantages of caching data to serve fast and highly scalable dynamic pages. So, let's take a look at the performance of cached and uncached articles now benchmarked with autobench and httperf:



Copyright © 1998-2003 Ace’s Hardware. All Rights Reserved.

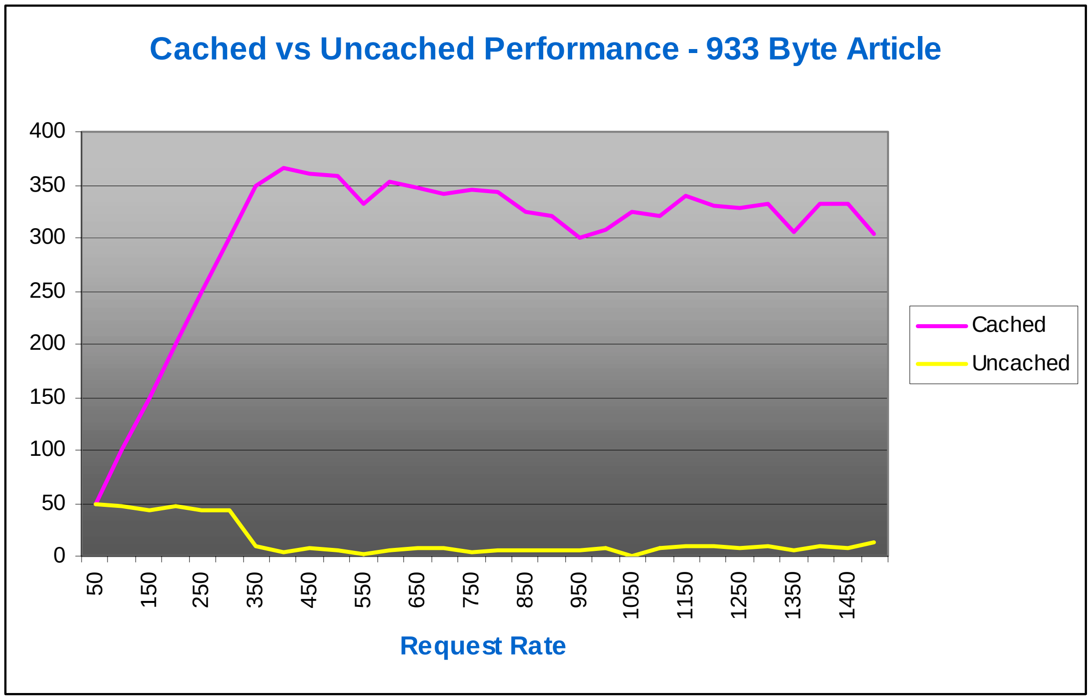
This is the same 18.8 KB article as the one that was benchmarked with ApacheBench on our production system. As was the case with those earlier benchmarks, the cached article code is identical to what is used on our server, while the uncached version has been modified to only read from the database.

The performance of the uncached version is quite dismal, reaching a maximum of 21.1 requests per second before the requests begin to pile up to the point where the server cannot return a response within the 5 second timeout imposed by the benchmark -- effectively reducing the server's output to zero. The cached version, on the other hand, delivers 50.2 requests per second at peak and generally fluctuates between 35 and 50 requests per second as the demand increases.

Why the poor performance for the uncached version? The database simply can't keep up. Since the article is uncached, each request means another query for the database, and the system cannot sustain 50 queries per second, let alone more. The maximum number of database connections was set to 75, but increasing this number further would not improve performance given how quickly the system was saturated.

Many high-load database driven sites implement a two or three tier model where the database runs on one server and the HTTP requests are handled by another (and the application on yet another in the case of a three tier model). This eliminates the competition for resources between these individual components that you might see on a single system, permitting a greater number of more complex database queries to be run, more complex pages to be generated, and so on. By having the application cache the data queried from the database, you can effectively eliminate the need to go to one of those tiers most of the time. This is exactly what you see for the cached performance result in the graph above.

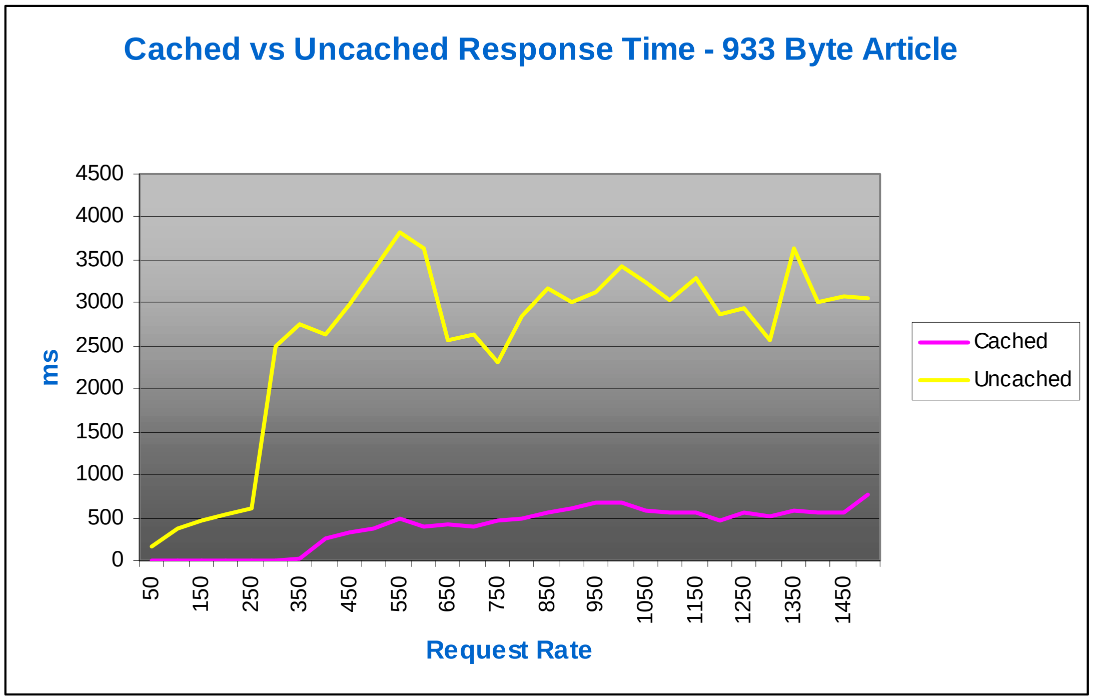
As we did earlier with ApacheBench, let's now take a look at the performance of the cache compared to the database by eliminating the article text from the actual HTTP output. The article text is still retrieved from where it is stored (the cache or the database) and is still formatted/processed as normal, but is not actually included in the output. As was the case before, the result is a 933 byte file that minimizes the HTTP workload relative to the application and database workloads.



This time around the database has a little more to work with, though the same holds true for the application in the case of the cached version. This translates into more performance, as the uncached version climbs to a peak of 48.2 requests per second and generally manages to maintain that same level of performance until the demanded request rate reaches 350 requests per second.

Copyright © 1998-2003 Ace's Hardware. All Rights Reserved.

The cached version remains significantly faster, however. It serves a maximum of 366.7 requests per second, and manages to sustain this level of performance for much longer than the database only version, despite increasing demand. Now let's compare the response times:



The response time is measured in milliseconds, so lower is better, and again the cached version clearly faster. While the best response time for the uncached version is 163.4 ms, that time climbs to almost 4 seconds (3815.6 ms) as the demanded request rate increases. The cached version starts with a response time of 3.1 ms at 50 requests per second, which increases to 382.2 ms at 500 requests per second, and finally 767.1 ms at 1500 requests per second.

Final Thoughts

There's always a lot of discussion about the newest CPUs, motherboards, and memory, but while thinking about all that performance, it may pay to spend a little time thinking about how to use it effectively. When you consider that the cached benchmarks performed in this article were anywhere from 2 to 7 times better than their uncached counterparts on the same hardware using a relatively simple in-memory caching algorithm that can be written in a matter of minutes for a persistent web application (like a Java servlet or JSP), you almost can't afford not to. To achieve the same level of performance through more traditional means might require an investment in significantly more expensive hardware and quite possibly a lot more of it. With this investment comes the potential for higher administration costs and requirements, more software licenses, and a generally more complex solution. With a bit of thought in the development, the web application we are running today is very simple, very scalable, very inexpensive, and very high performance.

Ultimately, scalability means getting the most out of your hardware. We've managed to extract quite a bit of performance from our current server, and yet there's still room to optimize even further. How much performance could we get out of a larger system, like a dual 2.8 GHz Pentium 4 Xeon, or an AMD Opteron-based server? That's a very good question. Until next time...

Copyright © 1998-2003 Ace’s Hardware. All Rights Reserved.