ScaleArc – Real-world Application Testing on WordPress

Peter Boros

April 29, 2014

ScaleArc: Real-world application testing with WordPress (benchmark test)

April 29, 2014 by Peter Boros

ScaleArc recently hired Percona to perform various tests on its database traffic management product. This post is the outcome of the benchmarks carried out by me and ScaleArc co-founder and chief architect, Uday Sawant.

The goal of this benchmark was to identify ScaleArc's overhead using a real-world application – the world's most popular *(according to wikipedia)* content management system and blog engine: WordPress.

The tests also sought to identify the benefit of caching for this type of workload. The caching parameters represent more real-life circumstances than we applied in the sysbench performance tests – the goal here was not just to saturate the cache. For this reason, we created an artificial WordPress blog with generated data.

The size of the database was roughly 4G. For this particular test, we saw that using ScaleArc introduces very little overhead and caching increased the throughput 3.5 times at peak capacity. In terms of response times, response times on queries for which we had a cache hit decreased substantially. For example, a 5-second main page load became less than 1 second when we had cache hits on certain queries. It's a bit hard to talk about response time here in general, because WordPress itself has different requests that are associated with different costs (computationally) and which have different response times.

Test description

The pre-generated test database contained the following:

- 100 users
- 25 categories
- 100.000 posts (stories)
- 300.000 comments (3 per post)

One iteration of the load contained the following:

- Homepage retrieval
- 10 story (post) page retrieval
- 3 category page retrieval
- Log in as a random user
- That random user posted a new story and commented on an existing post

We think that the usage pattern is close to reality – most people just visit blogs, but some write posts and comments. For the test, we used WordPress version 3.8.1. We wrote a simple shell script that could do these iterations using multiple processes. Some of this testing pattern, however, is not realistic. Some posts will always have many more comments than others, and some posts won't have any comments at all. This test doesn't take that nuance into account, but that doesn't change the big picture. Choosing a random post to comment on will give us a uniform comment distribution.

We measured 3 scenarios:

- Direct connection to the database (direct_wp).
- Connection through ScaleArc without caching.
- Connection through ScaleArc with caching enabled.

When caching is enabled, queries belonging to comments were cached for 5 minutes, queries belonging to the home page were cached for 15 minutes, and queries belonging to stories (posts) were cached for 30 minutes.

We varied the number of parallel iterations. Each test ran for an hour.

Results for direct database connection

```
Shell
Threads: 1, Iterations: 180, Ti
Threads: 2, Iterations: 356, Ti
Threads: 4, Iterations: 780, Ti
Threads: 8, Iterations: 1408, T
```

1	Threads: 1, Iterations: 180, Time[sec]: 3605
2	Threads: 2, Iterations: 356, Time[sec]: 3616
3	Threads: 4, Iterations: 780, Time[sec]: 3618
4	Threads: 8, Iterations: 1408, Time[sec]: 3614
5	Threads: 16, Iterations: 2144, Time[sec]: 3619
6	Threads: 32, Iterations: 2432, Time[sec]: 3646
7	Threads: 64, Iterations: 2368, Time[sec]: 3635
8	Threads: 128, Iterations: 2432, Time[sec]: 3722

The result above is the summary output of the script we used. The data shows we reach peak capacity at 32 concurrent threads.

Results for connecting through ScaleArc

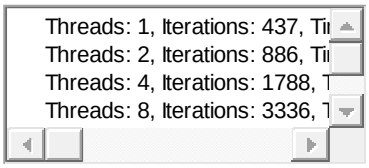
```
Shell
Threads: 1, Iterations: 171, Ti
Threads: 2, Iterations: 342, Ti
Threads: 4, Iterations: 740, Ti
Threads: 8, Iterations: 1304, T
```

```
1    Threads: 1, Iterations: 171, Time[sec]: 3604
2    Threads: 2, Iterations: 342, Time[sec]: 3606
3    Threads: 4, Iterations: 740, Time[sec]: 3619
4    Threads: 8, Iterations: 1304, Time[sec]: 3609
5    Threads: 16, Iterations: 2048, Time[sec]: 3625
6    Threads: 32, Iterations: 2336, Time[sec]: 3638
7    Threads: 64, Iterations: 2304, Time[sec]: 3678
8    Threads: 128, Iterations: 2304, Time[sec]: 3675
```

The results are almost identical. Because a typical query in this example is quite expensive, the overhead of ScaleArc here is barely measurable.

Results for connecting through ScaleArc with caching enabled

Shell

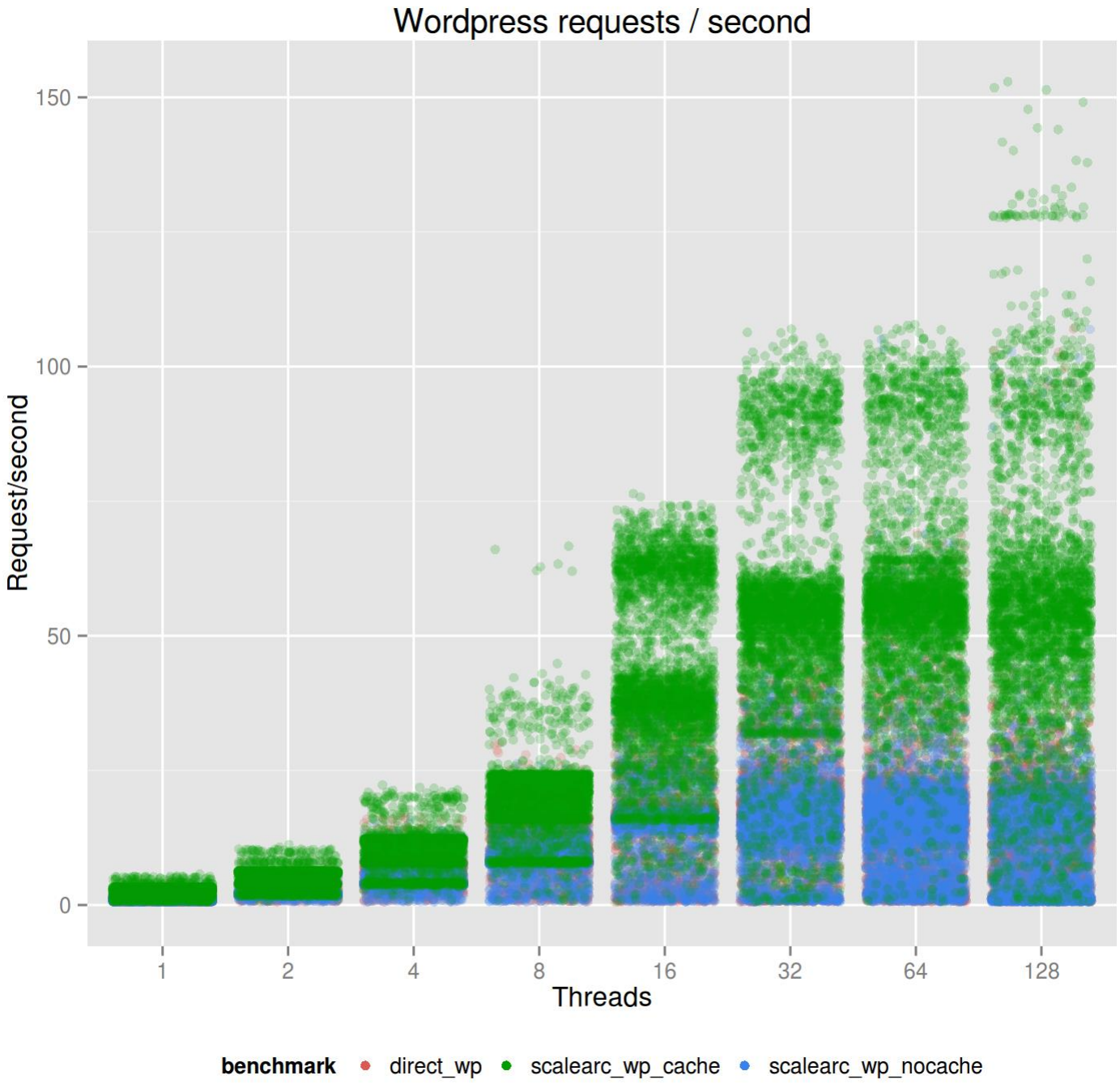



```
1    Threads: 1, Iterations: 437, Time[sec]: 3601
2    Threads: 2, Iterations: 886, Time[sec]: 3604
3    Threads: 4, Iterations: 1788, Time[sec]: 3605
4    Threads: 8, Iterations: 3336, Time[sec]: 3600
5    Threads: 16, Iterations: 6880, Time[sec]: 3606
6    Threads: 32, Iterations: 8832, Time[sec]: 3600
7    Threads: 64, Iterations: 9024, Time[sec]: 3614
8    Threads: 128, Iterations: 8576, Time[sec]: 3630
```

Caching improved response time even for a single thread. At 32 threads, we see more than 3.5x improvement in throughput. Caching is a great help here for the same reason the overhead is barely measurable: the queries are more expensive in general, so more resources are spared when they are not run.

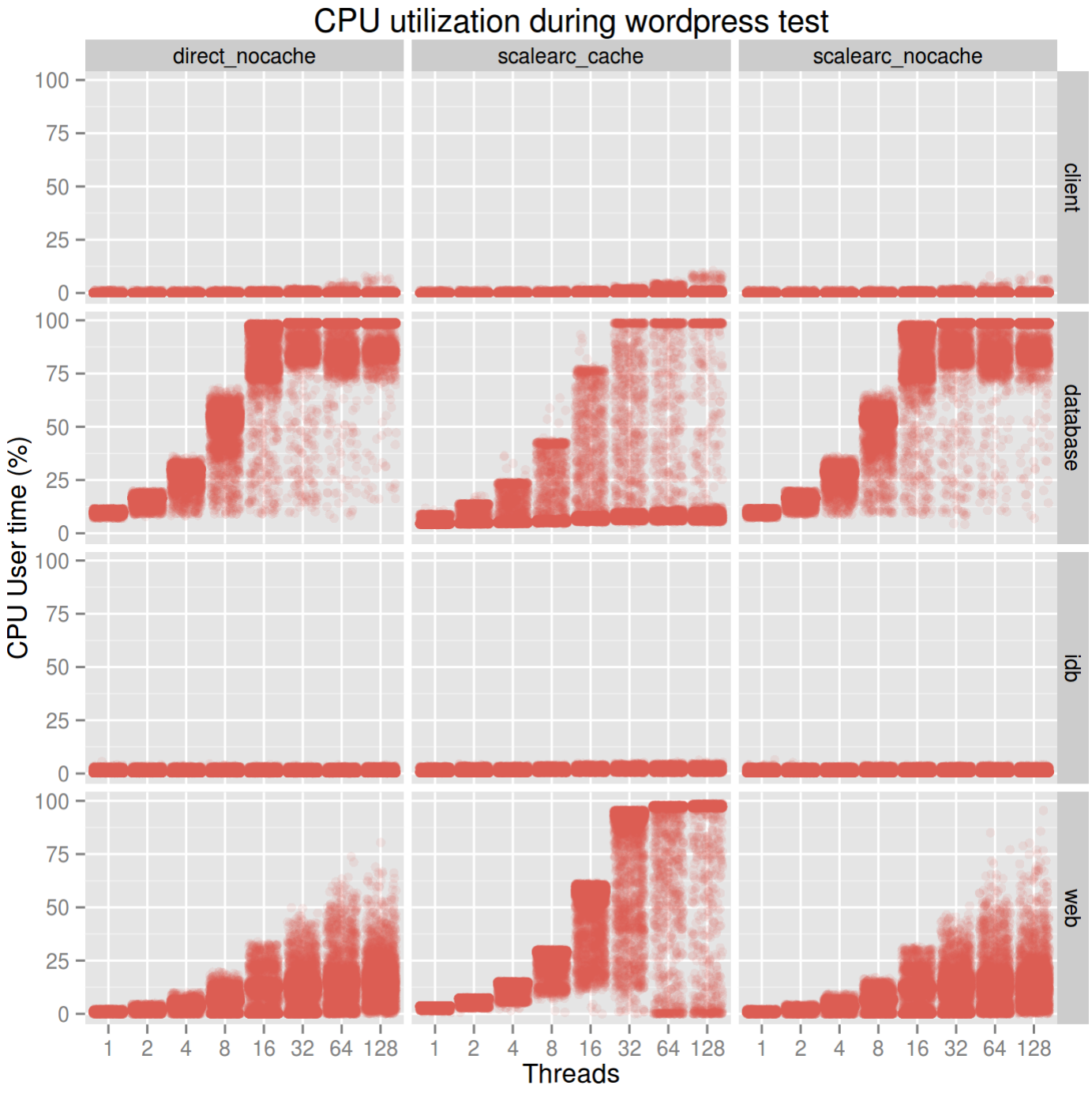
Throughput

From the web server's access log, we created a per-second throughput graph. We are talking about requests per second here. Please note that the variance is relatively high, because the requests are not identical – retrieving the main page is a different request and has a different cost then retrieving a story page.



The red and blue dots are literally plotted on top of each other – the green bar is always on top of them. The green ones have a greater variance because even though we had caching enabled during the test, we used more realistic TTLs in this cache, so cached items did actually expire during the test. When the cache was expired, requests took longer, so the throughput was lower. When the cache was populated, requests took a shorter amount of time, so the throughput was higher.

CPU utilization



CPU utilization characteristics are pretty much the same on the left and right sides (direct connection on the left and ScaleArc without caching on the right). In the middle, we can see that the web server's CPU gets completely utilized sooner with caching. Because data comes faster from the cache, it serves more requests, which costs more resources computationally. On the

other hand, the database server's CPU utilization is significantly lower when caching is used. The bar is on the top on the left and right sides – in the middle, we have bars both at the top and at the bottom. The test is utilizing the database server's CPU completely only when we hit cache misses.

Because ScaleArc serves the cache hits, and these requests are not hitting the database, the database is not used at all when requests are served from the cache. In the case of tests with caching on, the bottleneck became the web server, which is a component that is a lot easier to scale than the database.

There are two more key points to take away here. First, regardless of whether caching is turned on or off, this workload is not too much for ScaleArc. Second, the client we ran the measurement scripts on was not the bottleneck.

Conclusion

The goal of these benchmarks was to show that ScaleArc has very little overhead and that caching can be beneficial for a real-world application, which has a "read mostly" workload with relatively expensive reads (expensive means that network round trip is not a significant contributor in the read's response time). A blog is exactly that type – typically, more people are visiting than commenting. The test showed that ScaleArc is capable of supporting this scenario well, delivering 3.5x throughput at peak capacity. It's worth mentioning that if this system needs to be scaled, more web servers could be added as well as more read slaves. Those read slaves can take up read queries by a WordPress plugin which allows this, or by ScaleArc's read-write splitting facility (it threats autocommit selects as reads), in the later case, the caching benefit is present for the slaves as well.

Filed Under: Benchmarks, Insight for DBAs, MySQL Tagged With: caching, Peter Boros, ScaleArc, Uday Sawant, WordPress

About Peter Boros

Peter joined the European consulting team in May 2012. Before joining Percona, among many other things, he worked at Sun Microsystems, specialized there in performance tuning and was a DBA at Hungary's largest social networking site. He also taught many Oracle University MySQL courses. He has been using and working with open source software from early 2000s. Peter's first and foremost professional interest is performance tuning.

He currently lives in Budapest, Hungary with his wife and son.