# Stretching Java™ HTTP Servers in the multi-core world


## Abstract

Despite its importance, scalability of modern applications is poorly understood. On one hand, we have applications business-logic and workload keep growing in complexity. On the other hand, fast-paced time-to-market and cloud computing platforms have contributed to popularize the usage of languages based on complex runtime systems, being Java ™ one the most used.

Because no two applications are alike or use the runtime in exactly the same fashion, there is no guarantee that any single set of parameters will be perfectly suited for every pair. This reality highlights how important is to conduct focused performance evaluation and tunning at runtime level.

By running series of experiments, we found out that, with high statistical confidence, the average throughput per-core of the aforementioned service running on a virtual machine (VM) with 2 available cores is better than the same service running on a VM with 4 cores. We are well aware that these results ask for more investigation and answer those questions is definitely part of our future work.


## 1. Introduction

Web and other distributed software systems are increasingly being deployed to support key aspects of businesses including sales, customer relationship management (CRM), and data processing. Examples include: online shopping, processing insurance claims, and processing financial trades. As these businesses grow, the systems that support their functions also need to grow to support more users, process more data, or both. As they grow, it is important to maintain their performance (responsiveness or throughput), as it has direct impact on business objectives [Smith and Williams 2002].

For example, the highest priority for nearly all retailers is to drive revenueand turn a profit. So, could the effect of not considering the performance of the website be really that detrimental? The answer is a bold yes. Some examples are: i) lost in revenue led by website failures or slow down during peak times or ii) funds drained from important projects due to patchwork solutions by adding more hardware and software [1]



Yet, despite its importance, scalability of modern applications is poorly understood. On one hand, we have applications business-logic and workload keep growing in complexity. On the other hand, fast-paced time-to-market and cloud computing platforms have contributed to popularize the usage of languages based on complex runtime systems, being Java ™ one the most used.

In this context, JVM (Java Virtual Machine) comes with a set of default parameters, trying to provide adequate performance out of the box for the broadest range of application-execution resource pairs. However, because no two applications are alike or use the runtime in exactly the same fashion, there is no guarantee that any single set of parameters will be perfectly suited for every pair. This reality highlights how important is to conduct focused performance evaluation and tunning at runtime level. Classic modelling techniques do not consider runtime complexity (Williams and Smith 2004) and automatic tunning (Jayasena et al. 2015) remains offline and time consuming.

This paper presents a rather surprising result: with high probability, a highly-loaded no-op REST endpoint running on an asynchronous event-driven Java ™ HTTP server does scaleup linearly (much worse than that). By running series of experiments, we found out that, with high statistical confidence, the average throughput per-core of the aforementioned service running on a virtual machine (VM) with 2 available cores is better than the same service running on a VM with 4 cores. We are well aware that these results ask for more investigation and answer those questions is definitely part of our future work.

The remainder of the paper is structured into the following sections: Section 2 reviews related work in the domain of web server modeling and Java ™ auto-tuning. Section 3 describes the experimental methodology and set up. Section 4 presents results obtained and possible threats to validity. Finally, in Section 5, we present concluding remarks and next steps.

## 2. Related Work

Scalability is one of the most important quality attributes of today's software systems. Yet, despite its importance, scalability in applications is poorly understood. Williams and Smith (2004) reviewed four models of scalability - Linear scalability, Amdahl's law, Super-Serial Model and Gustafson's Law - and showed how they relate to Web and other distributed applications. The work also presents a pragmatic way of using well known models and regression to estimate capacity and find system bottlenecks but does not consider runtime systems as part of the complexity being modeled.

As modeling a complex system such as JVM, approaches like JVM auto-tuning have been increasing in popularity. Performance optimization in the context of high-end programs that consume a lot of system resources is very important. Jayasena et al. (2015) described HotSpot Auto-tuner, an offline, automatic tuner that considers the entire JVM and the effect of all the flags. Even though it is general-purpose and could lead to quite nice gains in performance, Auto-tuner tunning process is very time consuming (~200 minutes). Another major drawback of this approach is its offline nature, as the system itself and its load characteristics are constantly changing (and have impact on tunning).

We believe that better understanding the runtime system through analytically modeling could be mixed with application trial runs to help on achieve a fast, online tunning. Furthermore it would allow predictions, which is very important on capacity planning. This work only scratches the surface by showing how the JVM could impact on the overall server speedup. which are key

## 3. Methodology

On one hand performance has a direct and important impact on business objectives. On the other hand, every service has a unique set of requirements and often execute in resources which vary in vastly different ways. As an attempt to provide adequate performance out of the box for the broadest range of application-execution resource pairs, JVM (Java™ Virtual Machine) comes with a set of default parameters.

However, because no two applications are alike or use the runtime in exactly the same fashion, there is no guarantee that any single set of parameters will be perfectly suited for every pair. This reality highlights how important is to conduct focused performance evaluation at runtime level. Thus, our goal with this paper is to shed some light on this issue by assessing how the overhead that is not application-specific (i.e. HTTP request handling framework, JVM maintenance tasks) impacts an asynchronous event-driven HTTP server that runs under high utilization.

In other words, our main goal to find which configuration leads to the greatest saturation point[1]. To do so, clients[2] collect one piece of data: the number of requests processed per second (throughput, $T$). Based on this information, we can calculate an indirect metric:

$$avgT(n_{vCPUs}) = \frac{T}{n_{vCPUs}}$$

Where $avgT(n_{VCPUs})$ is the average throughput per available virtual CPU (VCPU), $n_{VCPUs}$ is the number of vCPUs available for execution and $T$ is the throughput. Based on the latter, we could to formally express the experiment hypotheses:

- $H_0$: The maximum throughput increases linearly with the increasing number of vCPUs allocated to run a highly-loaded single-endpoint REST service on an asynchronous event-driven Java™ HTTP server configured with default JVM flags. Or more formally:

$$max(avgT(k)) \approx max(avgT(k+1)),\ \forall k > 0$$

- $H_1$: The maximum throughput does not increase linearly with the increasing number of vCPUs allocated to run a highly-loaded single-endpoint REST service on an asynchronous event-driven Java ™ HTTP server configured with default JVM flags. Or more formally:

$$\exists k > 0,\ max(avgT(k)) \neq max(avgT(k+1))$$

### 3.1 Experimental Setup

To achieve the experiment goal, the Java HTTP server business logic overhead was trimmed down. The request handling internals was handled by i) Netty (Maurer and Wolfthal 2015), an asynchronous event-driven network application framework and ii) Jooby (The Jooby Project 2016), a micro web framework for Java, both without any custom parameters or configuration. The request handling logic was negligible, as well as the request and response payloads[3]. The Java HotSpot Virtual Machine was used in all experiments.

The load was impressed on the server using the monotonic step-function(Wikipedia 2016) described bellow. At every 10 seconds, the load was increased by:

$$load(i) = i \cdot n_{vCPUs} \cdot 50,\ \forall x, y,\ x \leq y \implies load(x) \leq load(y)$$

Where $i$ is the index of the current step. At the beginning of every experiment the server received a load of $n_{vCPUs} * 50$ QPS. This warm up phase aimed to mitigate the impact of Java's Just In Time (JIT) compiler and other internal JVM mechanisms.

We conducted a series of experiments varying the number of CPUs available at the virtual machine executing the server. Experiments were conducted using 1, 2 and 4 vCPUs (named *Exp1*, *Exp2*, *Exp4*). Each experiment was repeated 10 times. Table 1 presents details of each experiment configuration.

| Name | #vCPUs | RAM(GB) | Disk(GB) | #Clients |
|---|---|---|---|---|
| Exp1 | 1 | 0.50 | 10.00 | 1 |
| Exp2 | 2 | 4.00 | 60.00 | 3 |
| Exp4 | 4 | 8.00 | 90.00 | 6 |

**Table 1.** Experiment setup.

As there is a negligible amount of serial work done at request handling (business logic), one trying to predict the server performance based on either Amdahl's Law (using $\sigma \approx 0$), Gustafson's Law (using $\sigma'(1) \approx 0$) or Super-Serial Model (using $\sigma \approx 0$ and $\gamma \approx 0$) would expect a speedup very close to linear(Williams and Smith 2004). In other words, the degree of parallelism in the application is such that it could make (almost) full use of the additional resources provided by scaling.

[1] The saturation point tells that the service has reached its maximum capacity for the specific version (code), configuration and environment (Hare 2012).

[2] Code can be found at https://git.io/v6lCf.

[3] Code can be found at https://git.io/v64D7.

To our surprise, experiment results refute that hypothesis. These results are presented at next Section.

## 4. Experimental Results

### 4.1 Exp1: Running Server in a 1-vCPU VM

During this experiment, client(s) sent up to 1500 requests per second (RPS). For this particular experiment, throughput (T) and average throughput (avgT) are equal because the virtual machine used to run the server has only one vCPU.

Figure 1 presents the summary results of this experiment. Error bars throughout this article indicate a 95% confident interval for the mean. The confidence interval was computed using Bootstrap re-sampling (simulation-based statistical estimation technique (Efron and Tibshirani 1994)) with 10,000 trials because distributions were non-normal.

We could see a pretty much linear server throughput increase until 894.11 RPS, at which point it is being impressed a load of 1500 RPS. That is the saturation point and at this point the server is operating at its full capacity.

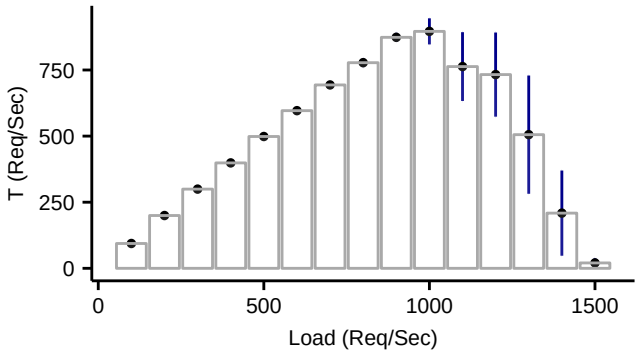


**Figure 1.** Throughput handled by the server given a certain load (1 vCPU).

After the saturation point the server is overloaded and as the load increases (simulating clients in a e-commerce website, for instance), the only effect is the growth of the server's accept queue (Banga and Druschel 1999). At some point requests are going to start to timeout and fail, leading to a decrease in throughput. Besides that, the runtime itself starts needed more bookkeeping, for instance, more stop-of-the-world pauses (Lo, Srisa-an, and Chang 2002) happen and more CPU is consumed by the garbage collector.

We believe all these reasons could explain:

- Steep decrease in the throughput: there is more and more competition for the only vCPU available
- Increase in the size of the error bars: due to the unpredictability nature of the GC and other runtime components leads to a great variance in the system performance

To better understand how these factors correlate is part our future plans. Finally, at the very end (1500 load) we have a practically inoperative server, successfully answering 20.4 RPS

### 4.2 Exp2: Running Server in a 2-vCPUs VM

During this experiment, the HTTP server was running in a virtual machine with 2 vCPUs. Again, jumping straight to the throughput distribution plot, which is presented in Figure 2, one could notice is the double size of vertical axis when compared to *Exp1* (1).

At this time, the server reaches saturation at 1950 RPS, handling 1751.27 requests per second on average. In other words, our expectations were confirmed: doubling the amount of resources from 1 to 2 vCPUs almost doubles the throughput (precisely 1.96).

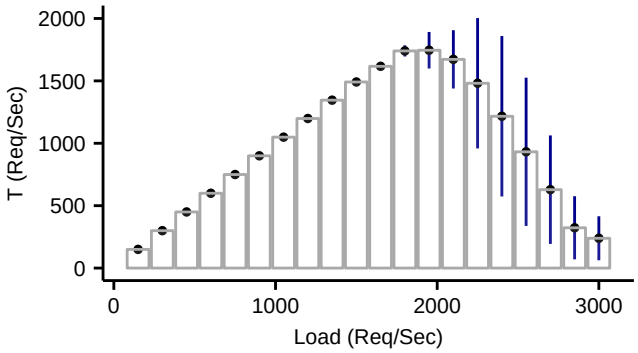


**Figure 2.** Throughput handled by the server given a certain load (2 vCPUs). Error bars indicate a 95% confident interval for the mean computed using Bootstrap resampling.

On the other extreme, at the right-hand side of the chart we see a stabilization trend around 259 RPS We need more experiments to increase our confidence, but it seems that adding the extra core increases the chance to the system to keep operating - even if in a very inefficient way.

Finally, the shape of the curve is also very similar to Figure 1 and we believe the steep decrease after saturation point and the size of the error bars are due to the same reasons described in *Exp1*. Figure 3 compares average throughput per core observed in *Exp1* and *Exp2*:

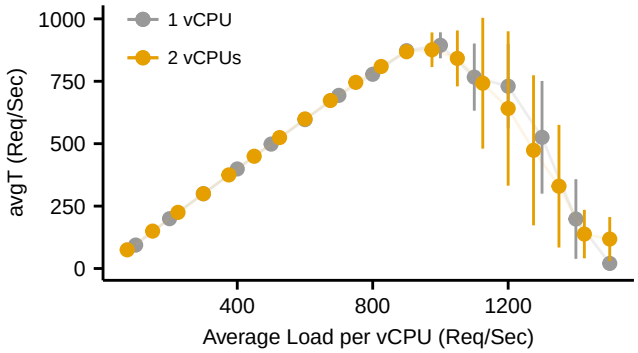


**Figure 3.** Comparing Exp1 and Exp2 avgT. Error bars indicate a 95% confident interval for the mean computed using Bootstrap resampling.

To be in pair with the averaged throughput, we averaged the load per vCPU as well. As one can see the, there is no statistical evidence of difference in the curves, which confirms our prior analysis.

### 4.3 Exp4: Running Server in a 4-vCPUs VM

In this case we have the HTTP server running in a virtual machine with 4 vCPUs. To our surprise, the maximum throughput handled by the server was only 1841.03 RPS. The peak throughput was reached under a load of 6000 RPS. Figure 4 shows distribution chart:

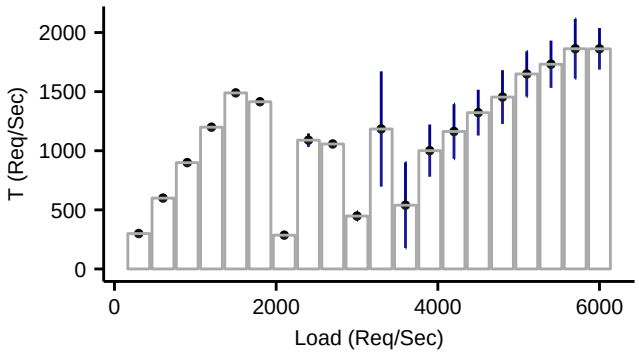


**Figure 4.** Throughput handled by the server given a certain load (4 vCPUs). Error bars indicate a 95% confident interval for the mean computed using Bootstrap resampling.

The first thing that calls out our attention is the bi-modal shape of the distribution. More importantly, looking at the confidence intervals, we could say this with quite high statistical confidence. That could be due to many reasons and investigating those reasons is definitely in our future work.

Figure 5 summarizes experiment results by comparing the average throughput versus average load per core in all cases. It provides us with visual evidence to refute the null hypothesis, as the average throughput of *Exp4* is far more worse than the other two cases.

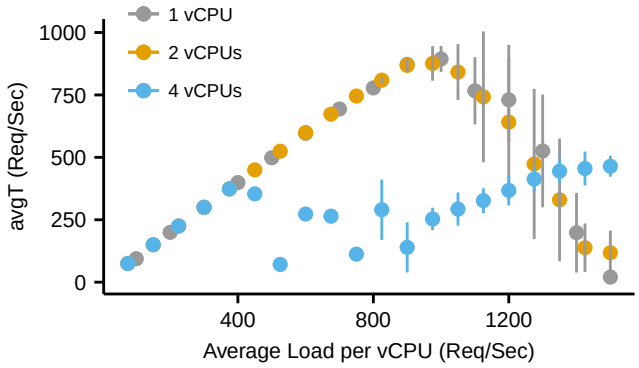


**Figure 5.** Comparing average troughput from servers running at VMs with 1, 2 and 4 vCPUs available. Error bars indicate a 95% confident interval for the mean computed using Bootstrap resampling.

To formally verify that, we calculated the 95% confident interval for mean throughput difference (*Exp4-Exp2*). We found that, with high probability, the mean difference is between -285.57 and -151.47 QPS. That result gives us enough confidence to reject our null hypotheses, as running a highly-loaded single-endpoint REST service on an asynchronous event-driven Java ™ HTTP server configured with default JVM flags in a 4 vCPUs' VM leads to a worse average throughput than 2 vCPUs.

### 4.4 Threats to Validity

Even though JVM is multi-platform, performance differences might arise when changing from one platform to the other. This work used only 64-bit Linux-based virtual machines. We also performed experiments only using Java HotSpot Virtual Machine

Other threats to external validity relate to the application and load characteristics. Even though the no-op request handling can be useful to isolate web framework and runtime overhead, it might not be representative. Same applying to the step-function used to generate load, which does not consider impact of burst traffic, for instance.

## 5. Conclusions and Next Steps

This paper presents a rather surprising result: with high probability, a highly-loaded no-op REST endpoint running on an asynchronous event-driven Java ™ HTTP server does scaleup linearly (much worse than that).

By running series of experiments, we have found that, with high statistical confidence, the average throughput per-core of the aforementioned service running on a virtual machine (VM) with 2 available cores is better than the same service running on a VM with 4 cores.

Our experiments also showed that, when running in VMs with one and two cores available, the server throughput decreases steeply after saturation. We believe this is because the JVM increases the competition for CPU to execute internal mechanisms such as garbage collection. This also reflects in a much bigger variance in the server performance.

Finally, another interesting result was that, when running in a VM with 4 cores, the server has a multi-modal distribution of the throughput. Understand this further is the bulk of our future work.

## References

Banga, Gaurav, and Peter Druschel. 1999. "Measuring the Capacity of a Web Server Under Realistic Loads." *World Wide Web* 2 (1-2). Hingham, MA, USA: Kluwer Academic Publishers: 69–83. doi:10.1023/A:1019292504731.

Efron, Bradley, and Robert J. Tibshirani. 1994. *An Introduction to the Bootstrap*. Chapman & Hall/CRC Monographs on Statistics & Applied Probability. Taylor & Francis. https://books.google.com.br/books?id=gLlpIUxRntoC.

Hare, David. 2012. "Performance Testing and Analysis with WebSphere Application Server." http://www.ibm.com/developerworks/websphere/techjournal/1208_hare/1208_hare.html.

Jayasena, Sanath, Milinda Fernando, Tharindu Rusira, Chalitha Perera, and Chamara Philips. 2015. "Auto-Tuning the Java Virtual Machine." In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop,*

*IPDPS 2015, Hyderabad, India, May 25-29, 2015*, 1261–70. doi:10.1109/IPDPSW.2015.84.

Lo, Chia-Tien Dan, Witawas Srisa-an, and J. Morris Chang. 2002. "A Performance Comparison Between Stop-the-World and Multithreaded Concurrent Generational Garbage Collection for Java." In *Performance, Computing, and Communications Conference, 2002. 21st IEEE International*, 301–8. doi:10.1109/IPCCC.2002.995163.

Maurer, Norman, and Marvin Allen Wolfthal. 2015. *Netty in Action*. 1st ed. Greenwich, CT, USA: Manning Publications Co.

The Jooby Project. 2016. "Jooby." Accessed August 17. http://jooby.org/.

Wikipedia. 2016. "Step Function." Accessed July 21. https://en.wikipedia.org/wiki/Step_function.

Williams, Lloyd G., and Connie U. Smith. 2004. "Web Application Scalability: A Model-Based Approach." In *30th International Computer Measurement Group Conference, December 5-10, 2004, Las Vegas, Nevada, USA, Proceedings*, 215–26. http://www.cmg.org/?s2member_file_download=/proceedings/2004/4246.pdf.