HHVM Performance Optimization for Large Scale Web Services

Yuhao Li
Meta Platforms
Menlo Park, CA, USA
lyuhao@meta.com

Abhishek Gupta
Meta Platforms
Menlo Park, CA, USA
abhigup@meta.com

Alex Yang
Meta Platforms
Menlo Park, CA, USA
alexkyang@meta.com

Peinan Chen
Meta Platforms
Menlo Park, CA, USA
peinanchen@meta.com

Joey Pinto
Meta Platforms
Menlo Park, CA, USA
pintojoey@meta.com

Brian Karrer
Meta Platforms
Menlo Park, CA, USA
briankarrer@meta.com

Mayank Pundir
Meta Platforms
Menlo Park, CA, USA
mpundir@meta.com

Maximilian Balandat
Meta Platforms
Menlo Park, CA, USA
balandat@meta.com

Arun Kejariwal
Meta Platforms
Menlo Park, CA, USA
akejariwal@meta.com

Benjamin Lee
University of Pennsylvania/Meta AI
Philadelphia, PA, USA
leebcc@seas.upenn.edu

ABSTRACT

HHVM is commonly developed for large online web services, yet there remains much room for optimizing HHVM performance. This paper discusses challenges and techniques in optimizing HHVM performance for Meta's web service. We begin by evaluating the effectiveness of semantic request routing, a request routing method aimed at enhancing code cache performance in HHVM, and examine its implications for optimizing HHVM performance. Second, we characterize HHVM performance for a large-scale datacenter and identify the challenges brought by uncontrollable confounding factors. Finally, we present the performance management framework for autotuning HHVM performance at scale.

CCS CONCEPTS

• **Information System** → **World Wide Web**; • **World Wide Web** → *Web Services*.

KEYWORDS

Web Service, HHVM, Performance Analysis, Performance Opitimization

ACM Reference Format:
Yuhao Li, Abhishek Gupta, Alex Yang, Peinan Chen, Joey Pinto, Brian Karrer, Mayank Pundir, Maximilian Balandat, Arun Kejariwal, and Benjamin Lee. 2023. HHVM Performance Optimization for Large Scale Web Services . In *Proceedings of the 2023 ACM/SPEC International Conference on Performance Engineering (ICPE '23), April 15–19, 2023, Coimbra, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3578244.3583720

ICPE '23, April 15–19, 2023, Coimbra, Portugal

ACM ISBN 979-8-4007-0068-2/23/04…$15.00
https://doi.org/10.1145/3578244.3583720

1 INTRODUCTION

HipHop Virtual Machine (HHVM) is a virtual machine designed for executing PHP and Hack codes, which are often used in web application development [43]. HHVM is widely used by large online service providers such as Meta[3], Baidu[4], Slack[6], *etc.* HHVM performance improvements are valuable and enhance the overall cost-efficiency of large-scale web services that support billions of users (*e.g.*, the Facebook app). A production web service can be deployed on hundreds of thousands of servers. On such a large scale, even a small percentage increase in efficiency could contribute to a significant reduction in capital and operational expenditures. This paper presents our HHVM optimization practice for web services at Meta. We discuss promising opportunities to optimize HHVM performance for large-scale web services. Moreover, we discuss unique challenges in measuring and optimizing HHVM performance at scale and present optimization frameworks to resolve these challenges.

First, opportunities exist in reducing the overheads of just-in-time (JIT) compilation. HHVM caches translated native machine codes in memory when compiling HipHop bytecode at runtime. Cached translations are re-used when encountering the same bytecodes in the future, avoiding dynamic compilation overheads [5]. 0n the other hand, we observe diverse PHP function calls in web requests, and such diversity could cause high code cache miss rates. Engineers and Researchers at Meta have designed a specialized scheduler that utilizes *semantic request routing* to improve code cache hit rates. Semantic request routing routes web requests from the same cluster to the same HHVM servers, thereby increasing

code cache hit rates and decreasing just-in-time compilation overheads. In this work, we conduct performance analysis to show that this practice reduces response latency and improves both CPU and memory utilization at scale.[1]

Second, opportunities arise in configuring the HHVM engine's thread groups. The HHVM engine deployed in Metais a complex process, comprising more than 70 thread groups serving different functionalities such as dynamic compilation, routing, and HTTP communication [9, 10, 43]. Each thread group has independently tunable properties such as thread pool size and affinity setting. However, most of these properties are set manually with engineering expertise and are often sub-optimal. Given these parameters, we propose an auto-tuning framework that configures threads to improve performance and efficiency for large-scale web services. We address these challenges with techniques in machine learning and experiment design [2] to explore the HHVM configuration space effectively.

Finally, we address difficulties in auotuning at scale due to noisy datacenters. Autotuning at the datacenter scale is non-trivial as numerous confounding factors across the system stack affect performance measurements. For example, these factors include randomness in the traffic load [38], variation in hardware manufacturing [37], and power capping decisions [49]. These factors, which impact performance differently across hosts and time, are impossible to control precisely. Performance variations may distort the auto-tuning framework's performance estimates for a given thread configuration, which in turn may lead to poor configuration decisions.Although software autotuning has been a focus in both industry and academia, how to autotune in a noisy execution environment is rarely discussed. Correspondingly, We present a characterization of the performance variation in the data center environment and discuss its impact on performance measurement and autotuning. Moreover,we address these challenges with A/B testing, which uses the concept of Randomized Controlled Trials [14] to control the effects of confounding factors and obtain a robust statistical estimate of performance. To sum up, this paper presents following contributions to HHVM optimization:

- **Semantic Request Routing Performance Analysis** We give an overview of semantic request routing as the background information, and present performance analysis of this technique and discuss its implication for HHVM performance tuning. Our evaluation indicates that such routing reduces request latency by 35% and reduces average CPU usage by 30%.
- **Understanding HHVM Performance Variation.** We characterize HHVM performance variations in a production datacenter and identify significant confounding factors that cause those variations.
- **Autotuning HHVM at Scale.** We present an HHVM autotuning framework that controls the effect of confounding factors while improving performance tuning speed and accuracy. The framework improves HHVM efficiency by 1% - 8%.

[1]Brian Karrer and Mayank Pundir contributed to the early research and development as well as initial performance tests of semantic routing. Joey Pinto contributed to the performance analysis of the latest semantic routing solution.

2 BACKGROUND

2.1 Web Services and HHVM

Meta deploys a large online service that receives millions of requests per second on a daily basis. It has developed a modular, service-based architecture to serve this large volume of user traffic [39, 46, 47]. The architecture includes load balancing, protocol routing (`Proxygen`[10]), request processing (`Web`[43]), key-value stores (`TAO`[18], `memcached`[41]), and database services.

In this paper, we focus on the `Web` service, which receives HTTP requests from the load balancer and responds with the desired content. This service is supported by hundreds of thousands of machines distributed across multiple regional datacenters. We seek to optimize service performance by targeting the HHVM engine at its core.

HHVM and Dynamic Compilation. HHVM is a virtual machine that efficiently executes PHP and Hack, the main programming languages for developing the web service studied in this paper. HHVM compiles PHP and Hack source codes into high-level bytecodes and uses just-in-time (JIT) compilation to compile necessary bytecodes into machine instructions for efficient execution dynamically.

When a request arrives, HHVM first identifies the PHP source files for the request and converts the PHP source into HipHop bytecode in the ahead-of-time stage [43]. Next, the bytecode is converted into native machine instructions for execution. HHVM first checks the code cache, which contains compiled machine instructions for previously executed bytecode. If the cache contains machine code for the target bytecode, HHVM executes that machine code directly. Otherwise, it uses the JIT compiler to generate the machine code.

Thread Composition in HHVM. HHVM comprises more than 70 thread groups. These thread groups have diverse functionalities and together handle the large request load received by each server. Furthermore, they can be categorized as foreground HHVM threads and background threads. Foreground threads are HHVM worker threads that receive HTTP requests, execute PHP functions, and send HTTP responses.

Background threads are non-PHP threads that execute other important tasks. Routing threads route requests to other threads for data, authentication, indexing, and consistency management. HTTP threads (*i.e.*, `proxygen` workers) maintain active sessions among services. Other thread groups route memory requests to in-memory caching services—`memcache`, `TAO` and `TACO`—and retrieve relevant data in response to user requests. The `memcache` service provides a general-purpose cache for small data objects. The `TAO` service is a high-performance service for caching and querying a graph of related and associated objects. The `TACO` service supports caching of ephemeral, local data.

Figure 1 breaks down CPU utilization into four major categories of thread groups. PHP threads consume most CPU cycles due to PHP's much larger thread pool. Although non-PHP threads consume 20% of CPU cycles, each non-PHP thread actually consumes more cycles than a PHP thread.

Figure 2 shows average CPU utilization per thread for the most compute-intensive thread groups. The `mcrpxy-tao` and `mcrpxy-web` thread groups, which routes requests to the `TAO` and `memcached`

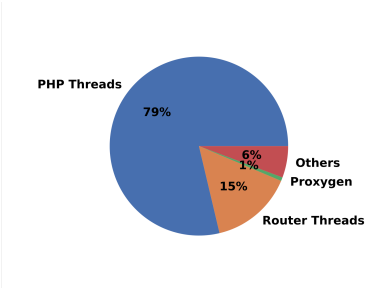


Figure 1: HHVM Processor Utilization. Pie chart breaks down processor cycles consumed by various thread groups.

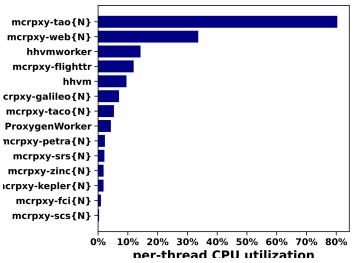


Figure 2: HHVM Processor Utilization. Bar chart indicates average processor cycles consumed per thread across various thread groups.

services, respectively, consume the most CPU cycles when measured on a per-thread basis. These compute-intensive threads are more likely to be performance bottlenecks that extend the critical path when handling a web request.

A number of configuration parameters can be used to adjust thread group performance. Each group has at least one tunable configuration parameter, the size of the thread pool. In addition, each group may have specific tunable properties. For example, HHVM router threads have an affinity option, which specifies whether a thread is pinned to a specific set of client servers. Pinning reduces the number of TCP connections a router needs to maintain and may improve the thread's memory efficiency.

Optimizing the thread configuration is challenging. Even if we were to restrict the optimization to thread pool size, we would need to consider that parameter for more than ten thread groups. Additional thread parameters would further increase the configuration space. Moreover, thread groups interact with each other and tuning each group independently is impossible. The large number of parameters and their interactions define a configuration space that cannot be optimized through exhaustive search.

2.2 Semantic Request Routing Design

HHVM caches machine instructions for future usage, but cache capacity is limited. Semantic request routing is a strategy for better using cache capacity, improving code cache performance, and improving JIT compilation efficiency. With semantic request routing, each web server specializes in handling a subset or partition of web requests. Within a subset, requests trigger the same PHP functions

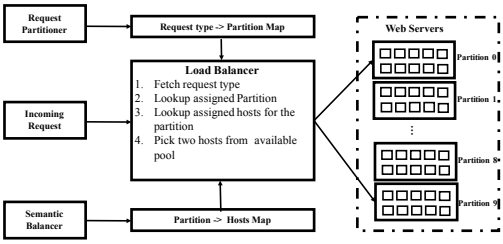


Figure 3: Semantic Routing Overview

and execute the same Hack codes. By limiting the diversity of code paths executed on each host server, semantic request routing improves HHVM code cache performance since each host is more likely to reuse already translated machine instructions.

Figure 3 outlines semantic request routing. A semantic partition is a logical container that mediates between request types and their assigned hosts. A **request partitioner** assigns each request type to a single semantic partition, and a partition can include several request types. The request partitioner uses a graph-cluster algorithm to partition requests. The algorithm starts by building a bipartite graph that has URL endpoints on one side (*e.g.*, /photos) and PHP functions on the other (*e.g.*, PhotoRender::getPhotoSize). A **semantic balancer** maps partitions to a subset of web servers responsible for serving requests from the corresponding partition,thereby mapping partitions to datacenter resources. The mapping seeks to balance load across partitions. If a partition's average load is greater or lower than the pool's average load, the balancer will add or remove host servers from the partition's allocation, respectively. The balancer adjusts the sizes for each semantic partition once every minute with a PID controller [24]. Currently, web requests are divided into 10 semantic partitions for the Meta's web service. This number is chosen based on experiments measuring the average working set size (number of function calls) among partitions: as the partition number increases beyond 10, the average working set size starts decreasing very slowly, and we receive a diminishing return from further increasing the partition number. Finally, the **load balancer**, given the request-to-partition mapping, identifies an incoming request's partition and then routes it to one of the hosts that support the corresponding partition. Host selection uses a "pick-2" algorithm [38]. This algorithm first selects two hosts from the pool corresponding to the partition at random. It compares loads on these two hosts and sends the request to the less loaded host. Note that the HHVM server is unaware of semantic request routing and can serve any request irrespective of the ideal partition.

3 SEMANTIC ROUTING PERFORMANCE AND IMPLICATION

We conduct a performance test to evaluate the benefits of semantic request routing. We select 1000 web servers in a regional datacenter and split them into two halves, one half with semantic request routing and the other half without. Finally, we direct the same production traffic to both sets of web servers.

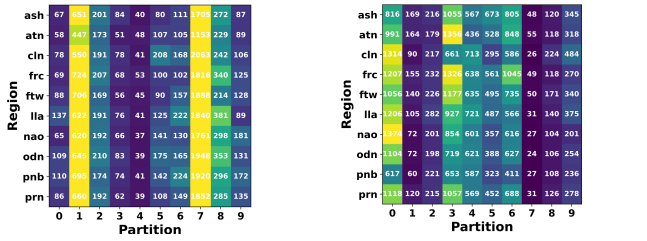
We observe significant improvement in two key performance metrics, HHVM memory usage and CPU utilization. First, semantic routing reduces request latency by up to 35%, due to less JIT compilation time in the request execution. Second, semantic routing reduces HHVM host memory usage by 5% on average. This reduction is a direct result of fewer PHP functions that must be translated and cached on the HHVM server. Finally, semantic routing improves processor utilization by 30%. HHVM spends less time on CPU-intensive instruction interpretation and more time on instruction execution.

We conduct separate performance analyses for each request partition, identifying diverse characteristics that motivate optimization strategies tailored to each partition. First, to understand the difference of requests across partitions, we profile request length and request load for each partition in 10 regional datacenters. Figures 4a–4b, for each partition and datacenter, present metrics averaged across host servers allocated for the partition. The data suggests differ significantly in request length and load. For example, requests in partition 0 require very little time to run those in partition 7 take up to seconds to complete. In contrast, the load in partition 0 is much higher than that in partition 7.

The diversity in request load motivates an HHVM engine with a unique thread configuration for each partition. One intuitive strategy configures a unique PHP thread pool size for each partition. The number of PHP threads should increase with the request load because it determines the maximum number of requests that the HHVM engine can serve in parallel. We adopt this idea to tune the number of PHP threads for each partition. But we leave the pool sizes for other thread groups largely unchanged and constant, as shown in Figure 5.

The performance characteristics of non-PHP threads (*i.e.*, background threads responsible for other functionalities) vary across partitions. We measure average CPU utilization per non-PHP thread and calculate a ratio of per-thread utilizations measured for non-PHP and PHP thread groups. The ratio characterizes each non-PHP thread group's load relative to the PHP thread group's load. A thread group that reports a higher ratio is more likely to be a bottleneck and on the critical path of execution within the HHVM engine. Figure 6 indicates that this ratio varies across partitions. For example, `mcrpxy-tao`'s ratio in partition 0 is 10 times that in partition 1. This profiling suggests that it is also necessary to configure non-PHP thread's configuration to optimize HHVM performance for each semantic partition.

Implications for AutoTuning. The complexity of tuning HHVM scales with the number of partitions. Furthermore, the tuning parameters interact with hardware factors. At Meta, we deploy four different server architectures to host web services [47]. Our exploration of the parameter space reveals that one particular set of HHVM thread parameters may benefit a partition on one architecture but harm the same partition on another. The ten partitions and four architectures lead to at least forty tuning and optimization problems. The scale of the problem motivates strategies that reduce the search space or explore a space more efficiently.



(a) request length (ms)

(b) request load (measured in request per second (RPS))

Figure 4: Varied request latency and throughput across partitions

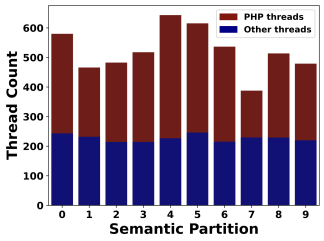


Figure 5: The numbers of PHP threads and non-PHP threads for each semantic partition.

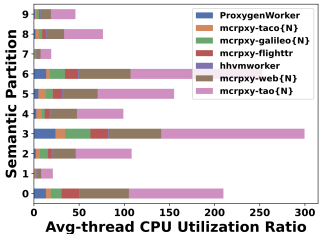


Figure 6: The ratio of average, per-thread processor utilization for non-PHP and PHP thread groups. Data presented for each semantic partition.

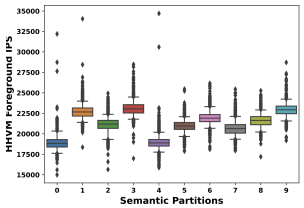


Figure 7: HHVM foreground performance (instructions per second) distribution for different semantic partitions.

4 PERFORMANCE VARIATIONS AND CONFOUNDERS

One of the most significant challenges to autotuning HHVM is variations in performance measurements. We observe consistent,

non-negligible variations in HHVM host performance in our datacenters. These variations introduce noise in performance profiles and complicates our efforts to interpret and tune HHVM performance. Therefore, we require strategies to distinguish between the deterministic performance impact from the configuration choice and random performance variations across host servers.

Profiling HHVM Performance. We characterize performance variations in production web fleets, profiling HHVM foreground performance (instructions per second, IPS) for each web host in a regional datacenter for a seven-day period. Our focus on HHVM foreground performance emphasizes a web host's HHVM worker threads and its ability to complete useful work in response to requests. The datacenter deploys thousands of web hosts. The load balancer allocates hosts to handle each web partition, assigning at least 800 hosts for each partition. The profiler measures HHVM foreground performance for each host every fifteen minutes, producing more than 800 time series for each partition. Each time series may have self-correlated variations due to diurnal web traffic patterns [33]. We take averages for these time series as the summary statistic for each host and then inspect the distribution of these averages across the hundreds of hosts assigned to the semantic partition.

Figure 7 uses the boxplot to show the distribution of hosts' HHVM foreground performance for ten semantic partitions. The standard deviation, which characterizes the population's variation in performance, is 3% 6% of the mean performance across different partitions; this statistic is often referred to as the coefficient of variation. Furthermore, the difference between upper and lower quantiles is significant; the differences between the third and first quartiles are more than 10% of the mean.

Confounding Factors. We study the possible causes of performance variations in the web fleet and identify web traffic volume (*i.e.*, request load directed to each web host) as a major confounder. In our datacenter, there exist persistent and significant variations in each host's traffic volumes even when they process requests from the same semantic partition. We observe a strong, linear correlation between the host's traffic volume and its HHVM foreground performance. For example, Figure 8 associates average request load with HHVM performance for hosts serving semantic partition 0. Individual web hosts show a wide range of request loads, which are associated with a wide range of HHVM foreground performance measurements.

Variations in request load is pervasive, appearing across regions and partitions. Table 1 reports the Coefficient of Variation (COV), which is defined as the standard deviation divided by mean, for hosts' average request load in a seven-day period across three datacenter regions. The coefficient of variation is 2% to 11% of the mean measurement. These variations make request load a critical confounding factor in configuration autotuning experiments.

Unfortunately, variation in request loads is hard to control and minimize. The variation is primarily due to random procedures in the web service load balancer. Web's load balancing algorithm uses a power-of-two-choices algorithm [38] for scalable, low-latency load balancing. When a request arrives, the power-of-two-choices technique randomly samples two hosts and selects the less-loaded host to handle the request. The randomness in the sampling procedures causes variation in each host's traffic.

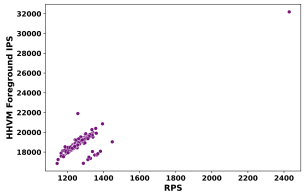


Figure 8: Request load (requests per second) versus HHVM foreground performance (instructions per second) for hosts serving semantic partition 0

Table 1: Coefficient of variation for host traffic, request load

Partition	Datacenter 1	Datacenter 2	Datacenter 3
0	7.0%	3.0%	11.3%
1	4.0%	4.6%	6.9%
2	4.0%	2.5%	8.1%
3	3.3%	2.9%	9.3%
4	4.5%	3.0%	8.4%
5	4.0%	2.9%	2.7%
6	3.8%	2.2%	3.6%
7	4.8%	2.4%	5.0%
8	3.7%	8.0%	9.0%
9	5.7%	2.7%	5.9%

We observe other confounding factors across the system stack that affect HHVM performance. These factors are hard to control and could cause spurious profiles of a candidate configuration when tuning performance. For example, we observe hardware variation in the fleet. Variations in the chip manufacturing process such that the same server architecture may operate at different processor and uncore frequencies. In addition, the placement of servers in datacenter aisles can affect cooling and lead to variations in server temperature. Power capping may impose varied power budgets across datacenter racks [29, 45, 49].

Implications for HHVM Thread Tuning. These variations impose significant challenges for configuration tuning. We often tune performance on small testbeds with a few HHVM hosts drawn from the whole datacenter. Our profiles indicate that more than 36% of machines report significant variations where performance is 1% higher/lower than the population mean. Suppose we profile a new configuration on the testbed and observe that its performance is 1% better than average performance in the datacenter. There is a 36% probability that this performance gain is due to inherent system variations rather than the new configuration.

Confounding factors, which are difficult to control during performance tuning, could produce imprecise or incorrect performance profiles for a candidate thread configuration. The autotuning framework must be aware of these confounding factors and design experiments to minimize the impact of potential confounding effects. Furthermore, the framework must provide meaningful confidence intervals for performance estimates conducted on small testbeds. The confidence interval represents the likely range of performance values at scale. To achieve these goals, we design and implement online, randomized controlled experiments.

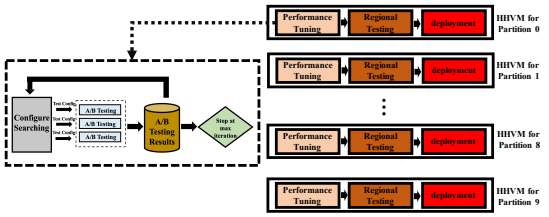


Figure 9: Configurating HHVM for Sematic Partitions

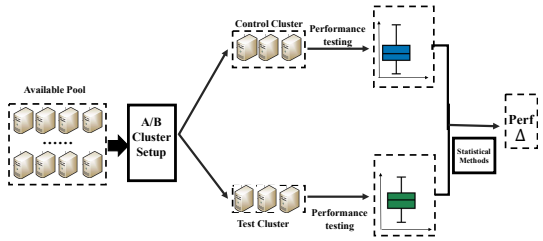


Figure 10: A/B Testing Procedure

5 AUTOTUNING FRAMEWORK

Figure 9 describes our framework for configuring and tuning HHVM thread groups for each semantic partition. The framework consists of three major steps for performance tuning, regional datacenter testing, and configuration deployment. Tuning refers to exploring a configuration space and evaluating candidate configurations on a testbed, defined by a subset of HHVM hosts in the datacenter. Testing refers to deploying a promising configuration on a selected regional datacenter to assess viability at scale. Finally, if no issues or challenges arise, a new configuration is scheduled for deployment across all datacenters.

In this section, we detail the performance tuning framework. It is aware of the datacenters' performance variation and efficiently identifies promising HHVM configurations while minimizing effects from confounding factors. Online randomized controlled experiments provide robust, unbiased performance estimates for candidate configurations by accounting for confounders. Using these robust estimates, search algorithms and heuristics explore the configuration space and identify promising configurations for deployment in datacenters.

5.1 Randomized Controlled Experiments

Online Randomized Controlled Experiment, also known as A/B testing [25, 27, 28], is a variant of Randomized Controlled Trials [14] used in clinical trials that study drug effectiveness. We use A/B testing to minimize confounding effects and obtain unbiased estimates of performance when profiling production systems. As shown in Figure 10, A/B testing creates two "statistically equivalent" clusters of web hosts,. One is the control cluster (A) and the other is the test cluster (B). The test cluster is assigned a candidate configuration to evaluate while the control cluster deploys the default, production configuration. A/B testing directs the same production traffic to both clusters. It then measures and compares performance statistics from the clusters. We estimate the performance difference between the A and B clusters, along with a confidence interval, to evaluate the candidate configuration.

Cluster Setup. First, we create A/B clusters with randomization, seeking to minimize the impact of hidden, uncontrolled factors or variables in the system. We select and specify the datacenter region, server types (*e.g.*, model, generation, kernel), and the number of servers (N) required for each cluster. We query the cluster management system to identify available host servers and select those matching the specified architecture. We select servers with the most common hardware and software configurations, reducing the potential bias from outlying platform characteristics. Specifically, we query and assess the host's processor temperature, server temperature, actual processor and uncore frequency, and actual number of active HHVM worker threads.

The cluster creation procedure identifies the servers whose key characteristics are close to the average for the server population. For each server, we calculate the difference between its measured metrics and the population's average metrics. We then determine the maximum difference across all metrics. We sort servers by their maximum difference and select the $2N$ that are closest to the mean. Finally, the procedure randomly splits these servers into two sets to form the test and control clusters.

Load Generation. The autotuning framework aims to drive servers, in both test and control clusters, to the maximum sustainable load in the production tier subject to a quality-of-service target. In our system, the load generator issues increases the number of requests issued to servers as long as average latency remains tolerable. A PID controller drives requests to machines and discovers load at which average request latency equals 100ms. The controller continuously monitors latency at each host and adjusts the control signal, which corresponds to the server's weight in the load balancer.

We stress the servers because HHVM instruction throughput at maximum sustainable loads permits more accurate and meaningful datacenter capacity planning. Datacenter operators seek cost efficiency by reducing the number of host servers required to serve a given number of web requests. The number of required servers is based on balancing peak demand from web users, measured in the number of HHVM instructions required to serve requests, and the sustainable supply from web servers, measured in instruction throughput at maximum sustainable load.

Cluster Comparison. Before we test two HHVM engine configurations for differences, we must ensure the control and test clusters that run those configurations are statistically indistinguishable and are not affected by confounders. In practice, we observe non-negligible variation across servers as load drivers determine maximum sustainable loads for each server. This variation is unavoidable due to inherent randomness in the load balancer, which uses probabilistic algorithms. The question is whether these statistical distribution of these variations suggest systemic differences between the servers in the control and test clusters.

We conduct A/A tests for the control and test cluster, checking whether the two clusters are statistically similar. The A/A test sets both control and test cluster to the default HHVM configuration. It performs load testing, collects performance metrics and compares

those metrics from the control and test clusters. The cluster setup is acceptable only if there is no significant difference between the control and test cluster. A significant difference between the clusters indicate confounders that impact performance and induce bias in one of the clusters. If such a difference exists, we repeat the server selection process to eliminate the difference. Otherwise, the server selection is successful, the cluster is acceptable, and we can proceed to A/B testing.

Statistically, A/A tests compare performance distributions from control and test clusters. We use the Student's t-test with a null hypothesis H_0 that states average performance from the test cluster equals average performance from the control cluster.

$$H_0 : \overline{p}_{\text{test}} = \overline{p}_{\text{ctrl}}$$

We use the T-statistic to compare the difference between $\overline{p}_{\text{test}}$ and $\overline{p}_{\text{ctrl}}$, accounting for variance in p_{test} and p_{control}:

$$T = \frac{\overline{p}_{\text{test}} - \overline{p}_{\text{ctrl}}}{\sqrt{\frac{var(p_{\text{test}})}{n_{\text{test}}} + \frac{var(p_{\text{ctrl}})}{n_{\text{ctrl}}}}}$$

The central limit theorem states that if H_0 is true, then the T-statistic should follow approximately a T distribution. Based on the T distribution, we calculate the p-value to measure the probability $P(T|H_0)$ of seeing T given H_0. If the p-value is small, say less than 0.05, we reject H_0 and conclude there is statistically significant difference between the mean of metrics in two clusters. Otherwise, we accept H_0 and conclude there is no statistically significant difference. We seek to accept H_0 for the test and control cluster before deploying and evaluating a candidate HHVM configuration.

Configuration Comparison. After the control and test clusters pass the A/A test, we compare a candidate HHVM configuration B against a default configuration A with an A/B test. The comparison is based on performance metrics collected on production traffic. Rather than compare pairwise performance for a request that executes on both clusters, we compare statistical performance distributions for the many requests that execute on both clusters. When we characterize performance distributions, we account for variations in measured system performance and ensure they do not skew conclusions about the two configurations' performance. For example, when we observe the performance difference between the test and control cluster is 1%, we must determine whether this 1% difference results from the candidate HHVM configuration or results from randomness in the system (*e.g.*, load balancing and the assignment of requests to servers).

We use statistical methods to rigorously detect the probability of a performance delta from a candidate, test configuration. We estimate the expected relative performance gain $\Delta = \frac{p_{\text{test}} - p_{\text{ctrl}}}{p_{\text{ctrl}}}$ at datacenter scale from sample performance data. Moreover, we obtain a confidence interval that might inform decision making and capacity planning at scale. The 95% confindence interval for Δ is estimated as follows where $T_{0.95}$ denotes the relevant statistic from the T distribution.

$$\left[\overline{\Delta} - T_{0.95} \cdot var(\Delta),\ \overline{\Delta} + T_{0.95} \cdot var(\Delta)\right]$$

We estimate $\overline{\Delta}$ with $\frac{\overline{p}_{\text{test}} - \overline{p}_{\text{ctrl}}}{\overline{p}_{\text{ctrl}}}$, which is a first-order Taylor series approximation of $\overline{\Delta}$. We can further improve the estimation by introducing higher-order terms[1]. Furthermore, we use the delta method to estimate $var(\Delta)$ as follows [27].

$$var\left(\frac{p_{test}}{p_{ctrl}}\right) = \frac{1}{\overline{p}^2_{\text{ctrl}}} \cdot var(p_{test}) + \frac{\overline{p}^2_{\text{test}}}{\overline{p}^4_{\text{ctrl}}} \cdot var(p_{ctrl})$$

Determining Testbed Size. The statistical procedures in our framework rely on Normality assumptions. When we compute p-values during A/A testing, we assume the t-statistic T approximately follows a Normal distribution. When we estimate the confidence interval for performance gains, we assume Δ follows a Normal distribution. In theory, according to the Central Limit Theorem, when we have enough data samples for the distributions we compare, both T and Δ follow asymptotic normal distributions.

In practice, we must determine the number of servers required for our testbed such that the Normality assumptions hold. We use a rule of thumb [27] to determine the minimum sample size to collect when conducting t-tests and calculating confidence intervals. This rule states we need at least $355 * S^2$ samples (i.e., servers) for the Normality assumption to hold where S is the distribution's skewness coefficient.

$$S(X) = \frac{E[X - E(X)]^3}{[Var(X)]^{3/2}}$$

Intuitively, we require more data to get a reliable confidence interval when the performance distribution is skewed. We determine the empirical skewness coefficient from profiled performance in production tiers.

5.2 Configuration Search

Around the A/B test framework, which statistically evaluates the performance of a candidate configuration, we develop a search procedure that determines the sequence of candidates to evaluate. This procedure explores, potentially in parallel, new configurations based on the results of previously evaluated configurations. The autotuning procedure iteratively selects new configurations, evaluates them with A/B testing, and then selects the next set of new configurations. Autotuning stops when a predetermined number of iterations or trials is reached.

We consider various auto-tuning strategies that vary in algorithmic complexity, tuning efficiency, and robustness to system uncertainty. We compare and contrast Bayesian Optimization (BO), Hill Climbing, Single Parameter Tuning, and Random Sampling. These algorithms have two significant advantages for tuning system performance. First, they permit black-box optimization and do not require an knowledge of the structure of the objective function, only that the objective can be evaluated. A black-box approach is vital for tuning web services as it is difficult or even intractable to model interactions between parameters and construct a useful analytic model. Second, these algorithms support parallel exploration of the configuration space. Parallelism is imperative for reducing experimental wall time because each A/B test requires hours of online profiling.

Bayesian Optimization (BO) is a machine learning technique particularly powerful for scenarios in which the evaluation of the objective is expensive and/or time-consuming. BO is an obvious

choice for HHVM tuning because A/B tests conduct long-running experiments.

BO consists of a surrogate model and an acquisition function. The surrogate model is a probabilistic model of the process to be optimized that can be evaluated much more cheaply or quickly than the true objective. A common choice for the surrogate model is a Gaussian process (GP), a non-parametric model known for providing well-calibrated uncertainty estimates. For any configuration x, it models expected performance $\mu(x)$ and uncertainty $\sigma(x)$ based on previously evaluated configurations and their A/B test data $\mathbf{D} = \{(x_1, \mu(x_1) \pm \sigma(x_1)),\ (x_2, \mu(x_2) \pm \sigma(x_2)),\ (x_3, \mu(x_3) \pm \sigma(x_3)),\ ...\}$.

The acquisition function, which operates on the model posterior, is optimized to suggest promising configurations for exploration. It balances exploitation (focusing on regions where mean performance is high) and exploration (focusing on regions where predictive uncertainty is high). Expected Improvement (EI), a popular acquisition function, evaluates the expected improvement of a configuration x over the best observed value so far, f^*, and has been shown to produce good, practical performance [15].

$$EI(x) = \mathbb{E}[max(f(x) - f^*, 0)]$$

The configurations with highest EI are selected (via numerical optimization) and evaluated on the true function in the next iteration. Initially, EI quickly explores the design space by collecting data for parts of the configuration space with relatively few observations and, as a result, have much higher predictive uncertainty. Once the configuration space is reasonably well-explored, EI naturally focuses on exploiting those configurations could most likely improve the current best configuration.

Note that, when using BO, we have the choice between sequential exploration, which only generates one new configuration to evaluate at each step, and parallel exploration, which generates a batch of new configurations to evaluate at each step. The sequential approach generally produces better optimization results as it utilizes strictly more information to acquire the same number of evaluations but is too time-consuming for the online A/B test.

We use open-source software Ax [2] to implement BO-based tuning. Ax provides several features that make BO better suited for our A/B tests at datacenter scale. It uses a variant of EI, Noisy EI [31], to account for noise in the observations as measured by the A/B testing setup. Moreover, it provides an efficient parallel BO implementation that balances end-to-end optimization time and quality of the optimized solution.

Hill Climbing (HC)[8] also performs search based optimization. HC starts with an arbitrary configuration and then seeks a better configuration by applying incremental changes to the configuration. If any incremental change leads to a better configuration, HC applies that change to the new configuration. HC continues this iterative procedure until no better configuration is found. We customize HC for our autotuning framework in several ways. First, we set the initial configuration as the default configuration used in the production tier. Second, we define the incremental change as the change that only updates one parameter in the configuration either upward or downward. For example, an incremental change for configuration $x = \{a_i, b_i, c_i...\}$ could be $x_{new} = \{a_{i-1}, b_i, c_i...\}$ and $x_{new} = \{a_{i+1}, b_i, c_i...\}$. HC explores all possible incremental changes and determines the best one with A/B testing. Third, when HC cannot find a better configuration through incremental updates, it randomly initiates an unexplored configuration as the starting point of a new search path.

Single Parameter tuning (SP) is another search-based tuning technique. For each parameter in the configuration, it explores all possible values of the parameter while fixing the other parameters at the default value. For a configuration space of dimension $D = \{N_1, N_2, N_3...N_i\}$, this method explores total $\Sigma_i N_i$ configurations. This method provides intuition for system architects as for each parameter which value is the best. However, this method fails to consider the interaction among parameters.

Random Sampling iteratively draws configurations, randomly and without repetition, for evaluation.

6 EVALUATION

The evaluation presents autotuning case studies for HHVM thread configuration. We evaluate appweight, a metric that evaluates web host capacity during A/B load tests, and describe how tuned HHVM configurations could lead to significant cost reductions at scale.

6.1 Experimental Methods

Semantic Partitions. We conduct HHVM autotune experiments for six semantic partitions—0, 2, 4, 6, 7, 8—in a regional datacenter. The partitions were selected for their diversity, request characteristics, and the number of hosts allocated for them. Partitions 0 and 7 have the shortest and longest requests, respectively. Partitions 2, 4, 6, and 8 are allocated the most servers.

Autotuning Cluster. Each partition is allocated five test clusters, each with 40 host, for a total of 200 servers to be used for autotuning experiments and A/B tests. Hosts are characterized and clusters are created to reduce systemic bias as discussed in the previous section. Multiple test clusters permit parallel exploration for new, candidate HHVM configurations. additional cluster is designated the control cluster and runs the default HHVM configuration.

Tunable Parameters. We apply autotuning to two significant and representative thread groups, `mcrpxy-TAO` and `mcrpxy-web`. These thread groups are among the most processor intensive and are most likely to benefit from HHVM performance gains. For both thread groups, we tune the number of threads in the pool. For `mcrpxy-TAO` we also tune thread affinity, a Boolean property that determines whether a routing thread should be bound to a subset of clients hosts. The binding could reduce the number of TCP connections (and overheads from maintaining those connections) between the web service and downstream cache services.

Table 2: Thread Configuration Parameters for Tuning

Parameter	Range
`mcrpxy-TAO` thread number	[2,11]
`mcrpxy-web` thread number	[2,11]
`mcrpxy-TAO` thread affinity	{True,False}

6.2 Appweight

Appweight is an internal metric that extends raw measures of HHVM instruction throughput. One of the key motivations for

improving HHVM performance is reducing the number of web hosts for user services and improving cost efficiency. However, raw HHVM instruction throughput is too granular a metric for capacity planning and cannot measure a server's capacity for serving web requests. Appweight addresses this challenge by directly estimating the number of web users that a given host can serve. This estimate then translates into the number of host servers required and allows datacenter operators to assess cost efficiency of the web tier.

$$\begin{aligned}\text{Appweight} &= \frac{\text{IPS}_{\text{web supply}}}{\text{IPS}_{\text{user demand}}} \\ &= \frac{(\text{HHVM foreground \%}) \times \text{IPS}_{\text{RCU}}/\text{RCU}}{\text{Total HHVM IPS/ALM}}\end{aligned}$$

Appweight is defined as the ratio between a web host's computational supply and web users' computational demands. The starting point is the server's maximum sustainable instruction throughput (IPS) given a 100ms latency target for web requests. Relative Compute Unit (RCU) is a normalized unit of server compute capacity. At Meta, the performance of each server type or generation is normalized to that of a baseline server architecture (*e.g.*, Intel's Nehalem). Thus, $\text{IPS}_{\text{RCU}}/\text{RCU}$ is normalized compute capacity per RCU. And $(\text{HHVM Foreground \%}) \times \text{IPS}_{\text{RCU}}/\text{RCU}$ is the RCU compute capacity for HHVM foreground threads that compute for meaningful user activities.

Active Last Minute (ALM) measures the number of unique users in a given minute. This metric is computed by sampling 1% of all hits to the production cluster. When planning server capacity, datacenter operators use ALM measured at the peak load, which is periodic and predictable. And Total HHVM IPS/ALM quantifies the average HHVM compute capacity demanded by a user.

The ratio estimates the max number of average users that can be supported by one RCU. As seen below, we can use this ratio to estimate how an improvement in Appweight reduces the number of RCUs required to serve user request. Supposing user demand is constant (*i.e.*, equation's right-hand side), then a 1% increase in Appweight permits a 1% reduction in the number of provisioned RCUs.

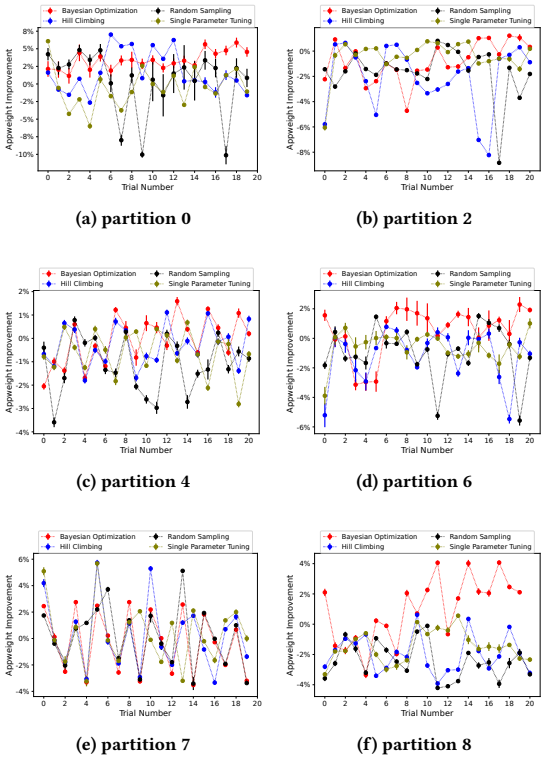
$$Appweight * RCU = \frac{(\text{HHVM foreground \%}) \times \text{IPS}_{RCU}}{\text{Total HHVM IPS/ALM}} \quad (1)$$

6.3 Autotune Efficiency

We experiment with four autotune methods, each running for five iterations. In each iteration, the algorithms generate four thread configurations for simultaneous A/B testing. Each test requires four hours to complete. In total, each method explores 20 configurations in the tuning procedure.

Figure 11 shows the Appweight improvement (larger is better) achieved by the configurations explored during autotuning. For each of six partitions, we evaluate the efficiency of the autotuning methods by comparing the best Appweight improvement across all A/B tests.

Single Parameter Tuning and Hill-Climbing tend to be more efficient given a smaller number of trials (*e.g.*, fewer than five). Both methods benefit from incrementally updating the configuration parameters, although the improvement is often less than 1%. In



(a) partition 0 (b) partition 2

(c) partition 4 (d) partition 6

(e) partition 7 (f) partition 8

Figure 11: Comparison between different autotune methods in tuning HHVM thread configurations for six semantic partitions. Circles represent the mean estimation of the performance difference between test and control clusters, and intervals around circles represent the variance estimation.

contrast, BO and random sampling suffer from exploring unknown regions in the configuration space.

Bayesian Optimization(BO) tends to find better configurations given a larger number of trails (*e.g.*, 15 to 20). The best configuration identified by BO outperforms the best of other methods by 0.8% - 3.0% with 20 trials. For most of the partitions, BO achieves better autotuning results after an adequate number of trials (e.g., 15 trials). For these partitions, BO builds a good Gaussian Process model after enough trials. Additionally, the acquisition function, an enhanced version of Expected Improvement, effectively guides the GP to exploit good configurations. One exception is at partition 7 (i.e.,Figure 12e), for which BO tends to degrade performance as more trials are conducted. We examine the original trace for partition 7 and look at configurations explored by BO autotuning in each trial. We find that BO starts to explore a new configuration region previously unexplored, after 15 trials. This new region consists of new combinations of `mcrpxy-TAO` and `mcrpxy-web` thread numbers which degrade HHVM performance. This happens when the BO's acquisition function suggests that such a new region is more valuable for exploration.

In sum, local search methods are preferable if the system's computational budget permits only a limited number of trials. If the

system's computational budget is more generous and permits a larger number of trials, BO likely discovers better configurations.

Preferred Thread Configurations. Our tuning experiments suggest that we cannot find a single set of thread numbers preferable to all six different semantic partitions. For example, we find that different partitions have distinct preferences for different `mcrpxy-TAO` thread numbers. Notably, partitions 2, 6 and 8 prefer a medium number (5 to 6) of `mcrpxy-TAO` threads; partitions 0 and 7 prefer the maximum number of `mcrpxy-TAO` threads available in the configuration space; and partition 4 prefers a relatively low `mcrpxy-TAO` thread (2-3). We argue that the best number of target threads depend not only on each partition's workload characteristics but also on each partition's own PHP thread pool size, as discussed in section 3. On the other hand, we discover that all semantic partitions prefer to turn on `mcrpxy-TAO` affinity. Turning on `mcrpxy-TAO` affinity can bind an `mcrpxy-TAO` thread to a specific subset of client hosts. It reduces the number of TCP connections between each web host and downstream services and improves efficiency.

6.4 Cost Efficiency Improvement

In summary, our autotuning improves appweights by 1%-8% across varied semantic partitions for the regional datacenter. In this subsection, we discuss the implications of the Appweight improvements brought by autotuning results.

From equation 1, a $1\% - 8\%$ increase in Appweight corresponds to a similar percentage of reduction in the computation units (i.e., web servers) required for serving these semantic partitions, provided that request loads from web users remain relatively constant. In the datacenter we study, each semantic partition is served by up to thousands of web servers. We estimate machines reduced for each semantic partition by examining appweight improvements in each individual semantic partition. Summing up the potential machine reduction across all semantic partitions, if we could reduce datacenter capacity by approximately five hundred machines, that could significantly reduce capital and operational expenditures for each regional datacenter, potentially saving millions of dollars if applied across many data centers.

7 RELATED WORK

HHVM Optimization. HHVM has many configuration parameters [7, 51], but there are few investigations on properly tuning these configuration parameters. Researchers have studied other aspects of HHVM performance including jump-start compilation[44] and architectural optimization [21]. Our work differs in that we optimize HHVM cache and thread group performance. Furthermore, we address challenges in noisy system measurements during performance optimization.

Partitions for Cloud Services. Partitioning is important when scaling out cloud services, dividing state or data into smaller segments to reduce per-machine data I/O and concurrent connections [11, 13]. For Partitioning and distributing data across machines is typical database services [12, 17, 40]. We introduce a new partitioning strategy that divides user requests instead of computational state. This idea is particular suitable for stateless micro services.

Performance Tuning. There is significant research in performance tuning for data center applications [34, 35, 53]. These works develop machine learning and search methods for tuning datacenter applications. To highlight a few of these efforts, SmartConfig applies control theory to performance-sensitive datacenter applications [48]. PARIS [50] and Selecta [23] use machine learning to tune virtual machine configurations for OLDI services and data analytics workloads, respectively. Sophia [35] and Optimus-Clouds [34] use predictive performance models to tune databases. However, few of these studies address issues in noisy performance measurement. Metis[32] uses adaptive re-sampling to address noise, identifying outlying performance measurements and re-sampling outliers to calculate confidence intervals. However, re-sampling cannot address confounding bias. We use randomized control experiments to eliminate confounding effects and obtain unbiased performance measurements.

Understanding and Mitigating Variance. Researchers have proposed varied methods to mitigate the impact of noisy systems on interactive datacenter applications. These methods include tuning hardware configurations [19, 22, 30], architectural optimizations [16, 36, 47] and resource contention management [20, 42, 52]. Sampling techniques are proposed to obtain a confidence bound on measurements for expected performance in a noisy system [32, 37]. These techniques calculate the number of repeated experiments for a desired confidence interval [37] or adaptively re-samples to remove performance outliers [32].

A/B Testing. A/B testing has been applied for online large scale web services [25, 26, 28] to understand the performance impact of new features. Metrics are collected from actions of web service users, which naturally constitute a large population and provide data samples. There is much less research describing A/B testing for tuning datacenter workloads, which must limit the amount of data to sample and collect because of constrained computational time and resources.

8 CONCLUSION

We describe practical techniques implemented to optimize HHVM performance for large-scale web services at Meta. First, the semantic request routing technique improves HHVM JIT efficiency by partitioning web requests by their PHP function calls and restricting each HHVM server to serve one request partition. Semantic request routing reduces HHVM request latency by 35% and improves HHVM CPU utilization by 30%. Second, we introduce our performance management framework for autotuning per-partition HHVM thread configuration. The framework utilizes A/B testing to rigorously measure performance in the noisy system. Furthermore, it applies advanced black-box tuning algorithms to explore configuration efficiently. Our result shows that our performance management framework can consistently improve HHVM cost-efficiency metric by 1% to 8% across different partitions.

9 ACKNOWLEDGEMENT

We acknowledge Alon Shalita, Igor Kabiljo, Andy Newell, Pol Mauri Luiz, Bin Liu, Max Wang, Yuliy Pisetsky, and Jo Bridgwater, who also and contributed to the design and continuous development of Semantic Routing, and provided valuable knowledge sharing.

REFERENCES

[1] Approximations for Mean and Variance of a Ratio. https://www.stat.cmu.edu/~hseltman/files/ratio.pdf, 2021. Online; accessed 12 Jan 2022.

[2] Ax:Adaptive Experimentation Platform. https://ax.dev/, 2021. Online; accessed 29 Nov 2021.

[3] Engineering at Meta: Redesigning the HHVM JIT compiler for better performance. engineering.fb.com/redesigning-the-hhvm-jit-compiler-\for-better-performance/, 2021. Online; accessed 29 Nov 2021.

[4] github:baidu-hhvm-301. https://github.com/baidu-lamp/baidu-hhvm-301, 2021. Online; accessed 2 Dec 2021.

[5] Go Faster! HHVM. https://hhvm.com/blog/4061/go-faster, 2021. Online; accessed 01 Jan 2022.

[6] Hacklang at Slack: A Better PHP. https://slack.engineering/hacklang-at-slack-a-better-php/, 2021. Online; accessed 2 Dec 2021.

[7] HHVM Configuration: INI Settings. https://docs.hhvm.com/hhvm/configuration/INI-settings, 2021. Online; accessed 2 Dec 2021.

[8] Hill climbing-Wikipedia. https://en.wikipedia.org/wiki/Hill_climbing, 2021. Online; accessed 12 Jan 2022.

[9] Introducing mcrouter: A memcached protocol router for scaling memcached deployments. https://engineering.fb.com/introducing-mcrouter-a-memcached-protocol-\router-for-scaling-memcached-deployments/, 2021. Online; accessed 29 Nov 2021.

[10] Introducing Proxygen, Facebook's C++ HTTP framework. https://engineering.fb.com/2014/11/05/production-engineering/introducing-proxygen-facebook-s-c-http-framework//, 2021. Online; accessed 1 Dec 2021.

[11] Partition around limits. https://docs.microsoft.com/en-us/azure/architecture/guide/design-principles/partition, 2021. Online; accessed 29 Nov 2021.

[12] Partition (database)–Wikipedia. https://en.wikipedia.org/wiki/Partition_(database), 2021. Online; accessed 27 Dec 2021.

[13] Partition service fabric reliable services. https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-concepts-partitioning, 2021. Online; accessed 27 Dec 2021.

[14] Randomized controlled trial. https://en.wikipedia.org/wiki/Randomized_controlled_trial, 2021. Online; accessed 29 Nov 2021.

[15] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, pages 469–482, 2017.

[16] Grant Ayers, Jung Ho Ahn, Christos Kozyrakis, and Parthasarathy Ranganathan. Memory hierarchy for web search. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 643–656. IEEE, 2018.

[17] Sikha Bagui and Loi Tang Nguyen. Database sharding: to provide fault tolerance and scalability of big data on the cloud. *International Journal of Cloud Applications and Computing (IJCAC)*, 5(2):36–52, 2015.

[18] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, et al. {TAO}: Facebook's distributed data store for the social graph. In *2013 {USENIX} Annual Technical Conference ({USENIX}{ATC} 13)*, pages 49–60, 2013.

[19] Kevin K Chang, Abhijith Kashyap, Hasan Hassan, Saugata Ghose, Kevin Hsieh, Donghyuk Lee, Tianshi Li, Gennady Pekhimenko, Samira Khan, and Onur Mutlu. Understanding latency variation in modern dram chips: Experimental characterization, analysis, and optimization. In *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science*, pages 323–336, 2016.

[20] Christina Delimitrou and Christos Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. *ACM SIGPLAN Notices*, 48(4):77–88, 2013.

[21] Dibakar Gope, David J Schlais, and Mikko H Lipasti. Architectural support for server-side php processing. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 507–520. IEEE, 2017.

[22] Chang-Hong Hsu, Yunqi Zhang, Michael A Laurenzano, David Meisner, Thomas Wenisch, Jason Mars, Lingjia Tang, and Ronald G Dreslinski. Adrenaline: Pinpointing and reining in tail queries with quick voltage boosting. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 271–282. IEEE, 2015.

[23] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. Selecta: Heterogeneous cloud storage configuration for data analytics. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 759–773, 2018.

[24] Carl Knospe. Pid control. *IEEE Control Systems Magazine*, 26(1):30–31, 2006.

[25] Ron Kohavi, Alex Deng, Brian Frasca, Toby Walker, Ya Xu, and Nils Pohlmann. Online controlled experiments at large scale. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1168–1176, 2013.

[26] Ron Kohavi, Roger Longbotham, Dan Sommerfield, and Randal M Henne. Controlled experiments on the web: survey and practical guide. *Data mining and knowledge discovery*, 18(1):140–181, 2009.

[27] Ron Kohavi, Diane Tang, and Ya Xu. *Trustworthy online controlled experiments: A practical guide to a/b testing*. Cambridge University Press, 2020.

[28] Ron Kohavi, Diane Tang, Ya Xu, Lars G Hemkens, and John PA Ioannidis. Online randomized controlled experiments at scale: lessons and extensions to medicine. *Trials*, 21(1):1–9, 2020.

[29] Vasileios Kontorinis, Liuyi Eric Zhang, Baris Aksanli, Jack Sampson, Houman Homayoun, Eddie Pettis, Dean M Tullsen, and Tajana Simunic Rosing. Managing distributed ups energy for effective power capping in data centers. In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, pages 488–499. IEEE, 2012.

[30] Rakesh Kumar, Boris Grot, and Vijay Nagarajan. Blasting through the front-end bottleneck with shotgun. *ACM SIGPLAN Notices*, 53(2):30–42, 2018.

[31] Benjamin Letham, Brian Karrer, Guilherme Ottoni, and Eytan Bakshy. Constrained bayesian optimization with noisy experiments. *Bayesian Analysis*, 14(2):495–519, 2019.

[32] Zhao Lucis Li, Chieh-Jan Mike Liang, Wenjia He, Lianjie Zhu, Wenjun Dai, Jin Jiang, and Guangzhong Sun. Metis: Robustly tuning tail latencies of cloud systems. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 981–992, 2018.

[33] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: Improving resource efficiency at scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 450–462, 2015.

[34] Ashraf Mahgoub, Alexander Michaelson Medoff, Rakesh Kumar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. {OPTIMUSCLOUD}: Heterogeneous configuration optimization for distributed databases in the cloud. In *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)*, pages 189–203, 2020.

[35] Ashraf Mahgoub, Paul Wood, Alexander Medoff, Subrata Mitra, Folker Meyer, Somali Chaterji, and Saurabh Bagchi. {SOPHIA}: Online reconfiguration of clustered nosql databases for time-varying workloads. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, pages 223–240, 2019.

[36] Hosein Mohammadi Makrani and Houman Homayoun. Mena: A memory navigator for modern hardware in a scale-out environment. In *2017 IEEE International Symposium on Workload Characterization (IISWC)*, pages 2–11. IEEE, 2017.

[37] Aleksander Maricq, Dmitry Duplyakin, Ivo Jimenez, Carlos Maltzahn, Ryan Stutsman, and Robert Ricci. Taming performance variability. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 409–425, 2018.

[38] Michael Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1094–1104, 2001.

[39] Usama Naseer, Luca Niccolini, Udip Pant, Alan Frindell, Ranjeeth Dasineni, and Theophilus A Benson. Zero downtime release: Disruption-free load balancing of a multi-billion user website. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 529–541, 2020.

[40] Shamkant Navathe, Stefano Ceri, Gio Wiederhold, and Jinglie Dou. Vertical partitioning algorithms for database design. *ACM Transactions on Database Systems (TODS)*, 9(4):680–710, 1984.

[41] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. Scaling memcache at facebook. In *10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*, pages 385–398, 2013.

[42] Rajiv Nishtala, Vinicius Petrucci, Paul Carpenter, and Magnus Sjalander. Twig: Multi-agent task management for colocated latency-critical cloud services. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 167–179. IEEE, 2020.

[43] Guilherme Ottoni. Hhvm jit: A profile-guided, region-based compiler for php and hack. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 151–165, 2018.

[44] Guilherme Ottoni and Bin Liu. Hhvm jump-start: Boosting both warmup and steady-state performance at scale. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 340–350. IEEE, 2021.

[45] Varun Sakalkar, Vasileios Kontorinis, David Landhuis, Shaohong Li, Darren De Ronde, Thomas Blooming, Anand Ramesh, James Kennedy, Christopher Malone, Jimmy Clidaras, et al. Data center power oversubscription with a medium voltage power plane and priority-aware capping. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 497–511, 2020.

[46] Akshitha Sriraman and Abhishek Dhanotia. Accelerometer: Understanding acceleration opportunities for data center overheads at hyperscale. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 733–750, 2020.

[47] Akshitha Sriraman, Abhishek Dhanotia, and Thomas F Wenisch. Softsku: Optimizing server architectures for microservice diversity@ scale. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 513–526, 2019.

[48] Shu Wang, Chi Li, Henry Hoffmann, Shan Lu, William Sentosa, and Achmad Imam Kistijantoro. Understanding and auto-adjusting performance-sensitive configurations. *ACM SIGPLAN Notices*, 53(2):154–168, 2018.

[49] Qiang Wu, Qingyuan Deng, Lakshmi Ganesh, Chang-Hong Hsu, Yun Jin, Sanjeev Kumar, Bin Li, Justin Meza, and Yee Jiun Song. Dynamo: Facebook's data center-wide power management system. *ACM SIGARCH Computer Architecture News*, 44(3):469–480, 2016.

[50] Neeraja J Yadwadkar, Bharath Hariharan, Joseph E Gonzalez, Burton Smith, and Randy H Katz. Selecting the best vm across multiple public clouds: A data-driven performance modeling approach. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 452–465, 2017.

[51] Owen Yamauchi. *Hack and HHVM: programming productivity without breaking things.* " O'Reilly Media, Inc.", 2015.

[52] Laiping Zhao, Yanan Yang, Kaixuan Zhang, Xiaobo Zhou, Tie Qiu, Keqiu Li, and Yungang Bao. Rhythm: component-distinguishable workload deployment in datacenters. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–17, 2020.

[53] Yuqing Zhu, Jianxun Liu, Mengying Guo, Yungang Bao, Wenlong Ma, Zhuoyue Liu, Kunpeng Song, and Yingchun Yang. Bestconfig: tapping the performance potential of systems via automatic configuration tuning. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 338–350, 2017.