DynaJET: Dynamic Java Efficiency Tuning

Karl Taht, Ivan Mitic, Adam Barth, Emilio Vecchio, Sameer Agarwal, Rajeev Balasubramonian, Ryan Stutsman

UUCS-20-001

School of Computing
University of Utah
Salt Lake City, UT 84112 USA

April 16, 2020

Abstract

Modern data analytics frameworks like Apache Spark are deployed at thousands of organizations regularly running diverse jobs that process petabytes of data. This has led to engineers carefully tuning configurations for these frameworks, as small percentages gains translate to large cost and energy savings at scale. However, this is challenging as frameworks and their runtimes can have hundreds of knobs, some with non-linear impacts on performance. Hence, tuning can require extensive end-to-end experimentation. This leaves administrators with a choice of performing extra offline work, or performing experimentation on real workloads at the risk of significant interim performance degradation.

We introduce a dynamic tuning approach, *DynaJET*, which offers several key benefits over prior approaches. The foremost improvement of DynaJET is operation at a much finer granularity. This enables testing and tuning experimentation on real workloads without the need for submitting separate jobs. In addition, tuning decisions are made in real-time, such that configuration choices can account for both current the compute task and current machine state. We demonstrate DynaJET's dynamic tuning atop the Spark compute engine using Java Virtual Machine (JVM) configuration parameters as an example. With just three JVM parameters, we find that a dynamic infrastructure can improve job completion time by up to 9.6% over a globally tuned JVM configuration, and at least 3.3% on average for an industrial data warehouse benchmark.

1 Introduction

Since the project's inception in 2009, Apache Spark has grown to be one of the most widely used big data analytics engines with more than 1,000 contributors to the open source project [1]. It is widely used not only at technology firms such as Facebook and Netflix, but also by banking, retail and academic infrastructures. Spark's popularity is due to its simultaneous generality and performance. In domain specific applications, Spark performs competitively with specialized solutions [2]. Yet, Spark provides a unified API with a number of built-in libraries that support key applications such as machine learning as well as stream and graph processing. Today, single Spark clusters exceed 8,000 nodes [3]. In short, Spark clusters run jobs that process petabytes of data each across thousands of machines and in thousands of organizations, so even modest, general gains in Spark performance can translate into large cost savings.

Spark performance was traditionally limited by network and disk bottlenecks [4]. However, with the release of Spark 2.0, significant optimizations to the compute engine mitigate unnecessary operations such as virtual function calls [5]. These optimizations have led to new bottlenecks: CPU and memory. Fundamentally, Java is in part to blame. For instance, Java's garbage collection incurs significant overhead compared to explicitly managed memory [6]. Newer versions of Spark use "off-heap" optimizations to mitigate this, but many similar JVM idiosyncrasies influence Spark performance.

Similarly, the Java JIT compiler dynamically interacts with the JVM at run-time to optimize code. However, the JIT compiler itself has more than 50 flags that affect performance and many of these have complex, non-linear effects. For example, function inlining generally improves performance until it hurts instruction cache hit rates, which can destroy performance. Ironically, many flags that can have drastic impact on performance are considered "Debugging Options", making these flags even more difficult to evaluate and interpret [7]. Spark workers run on the JVM, so a well-tuned JIT compiler is critical for Spark performance.

Users have two high level options when it comes to parameter tuning: hire a domain expert or use an automated black-box tuning framework. More recently, the latter has become more prevalent [8–12]. These frameworks typically frame the problem as follows: given a set of tuning knobs, design an automatic process that determines a configuration that optimizes average performance. Prior work has addressed how to determine the most important tuning knobs and how to select meaningful performance metrics. Other algorithms also detail methods for searching through the configuration space to converge to near-optimal solutions [8–10].

Automated approaches leave some open questions when translating solutions to a production environment. How much time should be invested in training and data collection versus deploying the solution? Training is fundamentally expensive, since it requires running a job from end-to-end in a test configuration. Is the environment in which training data was collected representative of the larger scale production environment? A fully loaded cluster may respond differently than a test suite with a long tail. And finally, how do we decide when to re-tune the parameters? Data, workloads, and even the systems themselves change gradually over time, meaning the optimization space is subject to concept drift. Therefore, we argue that parameter optimization should (i) include lightweight data collection, (ii) account for environmental status, and (iii) lend itself to continuous optimization.

To achieve these three goals, we present a new dynamic tuning approach, the Dynamic Java Effiency Tuner (DynaJET). This is made possible by Spark's unique architecture that deploys short-lived "workers", called executors, to process parts of a larger job. By tapping into individual executors, we enable fine-grain sampling and control over parameter tuning. This makes data collection of experimental parameters far cheaper than end-to-end performance metrics. Control at this layer also grants access to the current host state, enabling dynamic control that can actively respond to the environment. Finally, we develop a system architecture that uses this fine-grain control to continuously improve the tuning configuration, optimizing for job completion time (JCT).

In summary, our key contributions are as follows:

- We present a testing and tuning methodology that enables sampling at two to three orders of magnitude finer granularity than the prior state-of-the-art by modifying JVM parameters at the executor level.
- We introduce JVM configuration tuning as a function not only of workload and data, but also of current system state. Our executor level tuning controllers use this information to guide the decision of optimal configuration.
- We develop a backend infrastructure to support continual learning about the configuration space.
- We collaborate with an industrial data warehouse to measure performance improvements of our proof-of-concept in a realistic environment. We show an average of 2.3% JCT improvement over the baseline when tuning just one JVM parameter, and 3.3% JCT improvement when tuning three JVM parameters.

The remainder of our paper is organized as follows. In section 2 we provide background on Spark architecture and JVM tuning parameters. Next, we provide a detailed description of our system's architecture in section 3. In section 4 we review the methodology of our approach, including assumptions and limitations. We present the dynamic tuning results in section 5, including additional analysis of our dynamic controller algorithm. We provide a brief survey of related work in section 6 before concluding in section 7.

2 Background

2.1 JVM Tuning

Performance gains are ultimately tied to intelligently choosing configuration parameters. Prior work explored approaches for automatically extracting important configuration parameters. The goal of this work is to understand the benefits of a dynamic tuning framework, rather than to develop a new approach for intelligently ranking parameter importance. We choose specific JVM tuning parameters known by domain experts to have significant performance impacts, and we briefly discuss how they are relevant to a dynamic environment.

For the scope of our work, we make a distinction between garbage collection and code compilation tuning. Garbage collection (GC) and memory usage related parameters are known to have significant performance impacts [13]. However, tuning GC is often dangerous because even small changes in parameter settings can cause out-of-memory errors and can crash executors.

Therefore, we focus on parameters related to dynamic just-in-time (JIT) compilation. The Java HotSpot VM parameters change how JIT compilation works. We examine code inlining, an optimization that inlines small and frequently used method bodies, therefore reducing the overhead of method invocation. By default, the HotSpot VM will inline small methods, controlled by the `MaxInlineSize` parameter. This parameter controls the maximum byte code size of a function to be automatically inlined. However, large functions can be inlined if used frequently enough. This is controlled by the parameter `FreqInlineSize`, which again specifies the byte code size limit for methods that can be inlined. By default, the `FreqInlineSize` parameter is about 10 times larger than the `MaxInlineSizeParameter`. We also examine `ReservedCodeCacheSize`. The code cache stores JIT compiled code. When the code cache fills up, the JIT compiler no longer runs and therefore no longer optimizes. In this sense, the compiler can no longer inline frequently used functions because it has no more room to do so. The parameter settings covered in this work are shown in Table 1.

All of these parameters have important interactions among each other. Allowing more inlining can boost performance, but inlining too many functions makes the code size large. While code is relatively small compared to data, growing code size too much leads to cache misses. This is particularly relevant since the difference between an L1 and L2 cache hit is typically an order of magnitude [14]. Moreover, because L1 and L2 caches are effectively shared because of hyper-threading [15], interactions between threads change effective size. This is why we explore tuning these parameters dynamically, based on system load, and by extension, cache pressure.

2.2 Spark Architecture

In order to understand how we can introduce dynamic tuning, we need to examine the architecture of Spark itself. Spark uses a master-worker architecture. The master, or Spark driver, hosts the Spark context that manages internal services including data management and scheduling (or interacting with a scheduling service), for example [16]. At a high level, the master is responsible for dividing the job into units of work called "tasks" and assigning them to executors.

However, in reality there are more layers of abstraction. In a typical instance of Spark, the driver will interact with cluster managers. These cluster managers then create executors and assign tasks. One particularly important feature of Spark is the dynamic allocation of executors [17]. In this mode, executors can be created or destroyed dynamically to meet the needs of the workload. The rationale is that an executor consumes resources regardless of whether or not it is actively processing a task. For example, most Spark cluster managers are configured to only allow a fixed number of executors per physical host, so even an inactive executor blocks physical CPU cores. By removing executors that are not actively processing, we can free up extraneous resources.

Spark can be configured to collect performance statistics at the granularity of per-executor. In the dynamic executor model, executors may have a life span ranging from tens of seconds to tens of minutes. In either case, this is typically shorter than the query as a whole. Moreover, since each executor runs in its own process, this allows for dynamic control of JVM parameters throughout the execution of a Spark job. Moreover, this is particularly relevant since not all Spark executors are created in the context of the same environment. Some executors will run on fully-loaded machines with a number of queued tasks and highly contended resources. Other executors will run at a time when very few resources are being used. Our proposed dynamic tuning architecture provides a way to simultaneously monitor the run time environment and configure JVM tuning parameters accordingly.

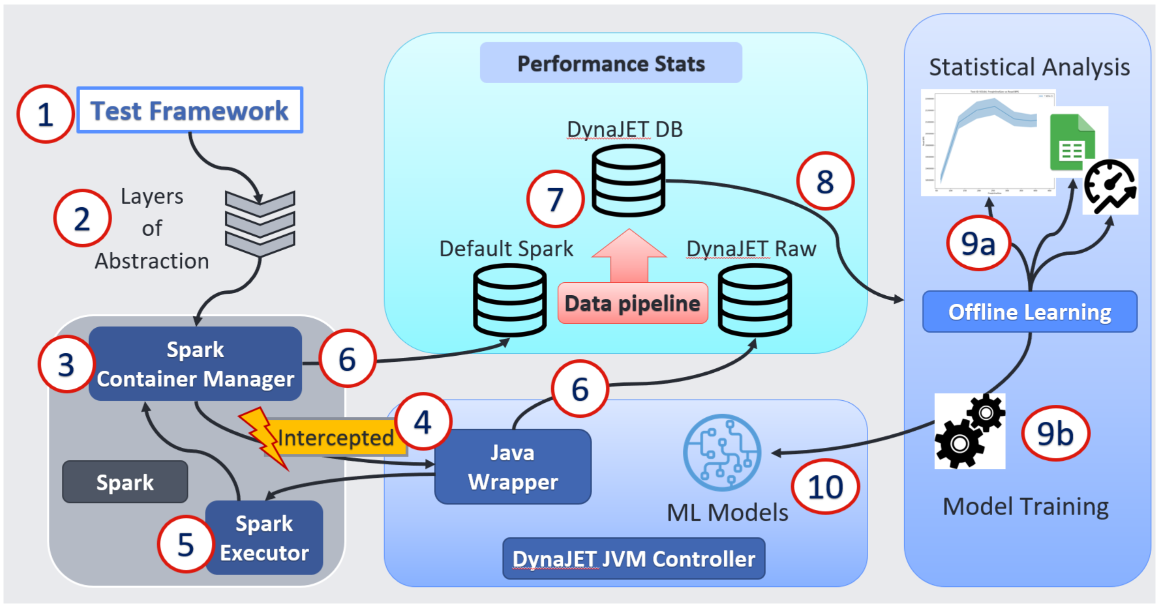


Figure 1: Graphical overview of the DynaJET Architecture. The details of each component and dataflow are described in detail in Section 3.

3 System Architecture

We now describe our dynamic tuning architecture for JVM tuning for Spark, DynaJET. Conceptually, the infrastructure comprises of two main pieces of software: an online layer that controls parameters and measures performance at the executor level, and an offline layer that analyzes the executor performance and trains models that are used to control parameters. Because of the interdependence of these two components, we discuss the framework in the order of how data flows through the framework. An overview of the architecture is presented in Figure 1, labeled in the order in which we discuss components of the framework.

3.1 The JET Dataflow

3.1.1 Running Queries

Any Spark job originates with submitting a program or query. We utilize an internal testing tool that automates running predefined queries. In practice, the tool is used both for correctness and performance testing. Performance was reported by running queries end-to-end and measuring total CPU time. However, because our infrastructure allows us to report fine-grained throughput statistics, we augmented the tool in several ways.

- *Continuous Loading Support.* We modified the tester to support continuously loading the cluster. Scheduling batches of test queries can result in artificial tails of under utilization when completing the last few queries in a batch. By continuously loading the cluster, we ensure the testing is more representative of a production environment with constant load.
- *Metadata Transfer.* By default, Spark executors have limited notion of what specific query they are running, only the task to perform. We embed metadata such as the query ID and run ID into the executor's environment so it can be picked up and used later by our JVM configuration controller.
- *Controller Parameters.* In addition to metadata, we also need to pass parameters JVM tuning controller. In a similar fashion to metadata, we pass "flags" to the JVM tuning controller via environment variables.

3.1.2 Layers of Abstraction

As noted in Section 2.2, the Spark architecture comprises of various levels of abstraction. The metadata encoded in the environment propagates through various layers of the compute stack.

3.1.3 Spark Container Manager and Interception

Finally, a Spark Container manager spawns a Spark executor to start assigning work in the form of tasks. We assign the Spark container manager's environment to point to our Java wrapper instead of the default `JAVA_HOME` location. This causes the Spark Container manager to call our wrapper instead of the JVM directly.

3.1.4 JVM Controller: Java Wrapper

The Java Wrapper is the transparent layer that exists between the Spark container manager and executors. Once activated, the Java wrapper first collects telemetry data of the system.

This is done by tapping into built-in Linux features that allow us to analyze the current system and memory load, captured as a percentage. We also use historical load via the number of queued processes in the past 1, 5, and 10 minute marks. In total, there are 5 input features collected. Once we have initial telemetry data about the system state, we can then choose how to set the JVM parameters. How this choice is made is described in section 3.1.8.

3.1.5 Creating the Spark Executor

Once the JVM parameters are chosen, the Java wrapper spawns the JVM for the Spark executor. The Java Wrapper attaches an instance of Linux Perf to the Spark executor process [18]. This enables us to collect hardware statistics at the executor level, such as instruction and cycle counts, as well as cache and branch misses.

3.1.6 Logging Metrics

Once an executor completes, Spark and the Java wrapper simultaneously log performance statistics to independent databases. Since the typical Spark job has in the order of hundreds of executors, we will log hundreds of individual samples for a single run. Note that each sample could be run with any JVM configuration, which allows us to sample alternate configurations much more rapidly than an end-to-end approach. Fundamentally, *this is why we achieve two to three orders of magnitude more data than prior tuning approaches*.

3.1.7 Data Aggregation

By default, each sample we log to the database contains some metadata about where the sample was collected from, initial system state information, and hardware statistics collected via Linux Perf. Initially, it may seem like this is enough information to proceed with optimizing the system, as we have a notion of throughput: instructions per cycle (IPC). Unfortunately, this does not hold true when it comes to tuning JVM parameters. By tuning parameters related to JIT compilation, we actually change the total number of instructions executed. Therefore, we need to use performance metrics collected at the software level.

Fortunately, the Spark engine reports of a number of statistics for individual executors that it logs to a separate metrics database. A natural question might be why not just use the

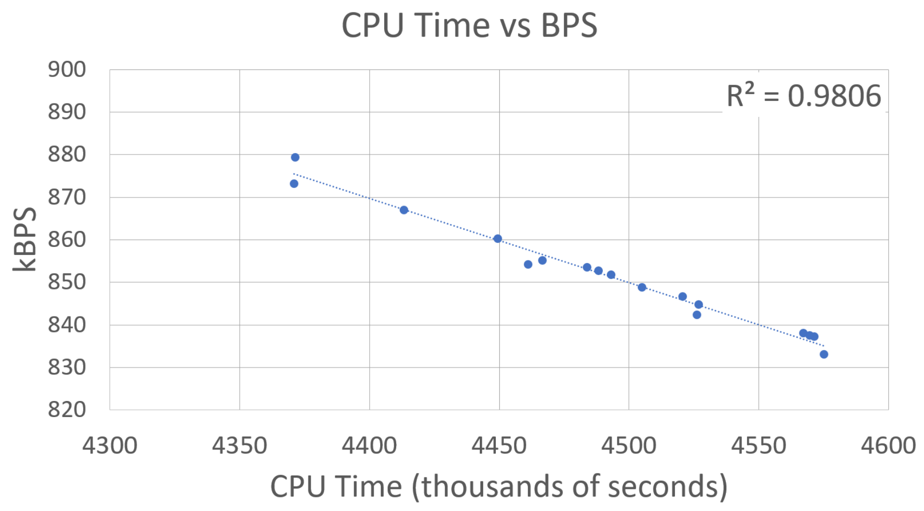
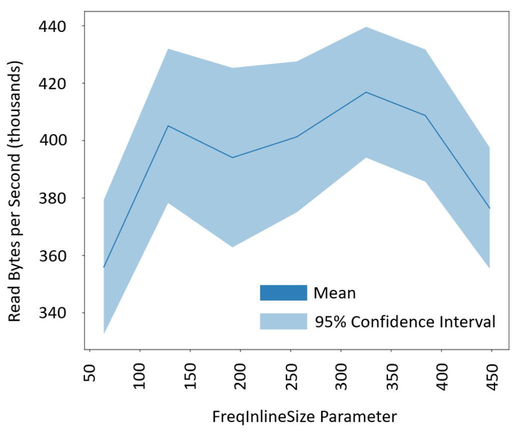


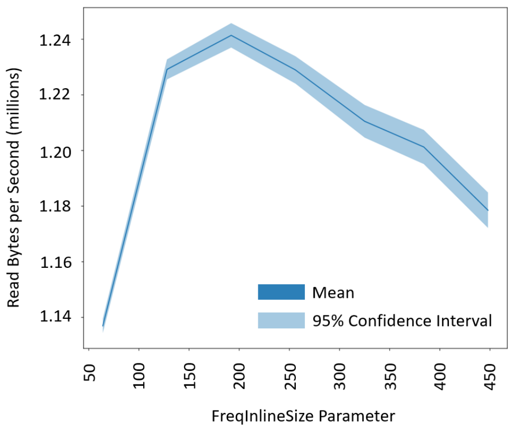
Figure 2: Scatter plot of 17 different runs of the same query plotting average throughput in kilobytes per second (kBPS) against JCT (CPU time). The nearly perfect linear relationship between BPS and CPU time validates that optimizing executors for BPS should optimize overall run time.

metrics reported by Spark then? First, we need to correlate host states with individual executors, which Spark does not capture. Second, the metrics reported by Spark do not contain sufficient metadata to know what query and what test run an executor's stats come from. We therefore build a data pipeline that joins the statistics captured by Spark and the Java wrapper to analyze per-query throughput without running them as individual test runs.

We use the number of read bytes as a metric to measure performance. Every query in our test suite is set to read a fixed number of bytes, thus, knowing how many bytes were read is proportional to how much meaningful work was done. Since we also know the amount of time each executor spent executing, bytes per second (BPS) gives us a meaningful throughput metric. We further verify this by comparing the average BPS, our throughput metric, to total CPU time. As shown in Figure 2, there is a linear relationship between improving BPS and reducing total CPU time. Thus, optimizing for this throughput metric enables us to optimize for JCT. We create a data pipeline that joins the Java wrapper's statistics with the bytes read reported by Spark for every executor. This aggregated database is labeled as the *DynaJET DB*.



(a) Query with only 320 samples



(b) Query with over 6000 samples

Figure 3: Both graphs plot the mean performance in seven different `FreqInlineSize` parameter values, with the confidence intervals plotted in light blue. While the left graph 45 samples per configuration, it fails to tightly bound expected performance. By comparison, the query on the right demonstrates the effects of having 20$\times$ the number of samples, resulting in much tighter confidence intervals.

3.1.8 Offline Learning

We now examine the backend learning framework for DynaJET. We opt to train individual controllers for each query. Query execution bottlenecks are often defined by the data it is processing, so training per-query allows us to incorporate that aspect. At first glance, this may seem like an unreasonable choice, but in fact, is highly applicable to a realistic data center environment. While other interactive compute engines, such as Presto, may run a query only one time, Spark is typically used for much larger tasks that will be run many times. Typically, data scientists will write large queries that become part of a data pipeline. These pipelines are typically run on a regular recurring schedule such as hourly, daily, or weekly for example [19]. Thus, it is reasonable for us to train individual models for important queries. In either the statistical or machine learning approach (described next), after training, the models are persisted as part of the Java wrapper package.

9a) Statistical Approach

As a first order, our approach includes tooling to perform statistical analysis using the fine-grain samples collected. By querying the DynaJET database, we can aggregate all data relating to a specific query. From there, we determine the average performance of each configuration tested. Note that not all executors perform equal amounts of work, thus they can't all be weighted equally. To this end, we compute the performance in a particular configuration as follows:

$$BPS_{avg} = \frac{\sum_i^N bytes}{\sum_i^N seconds} \neq \frac{1}{N} \sum_i^N \frac{bytes}{seconds}$$

This formulation ensures that we choose a configuration that is most efficient at processing bytes overall. However, we supplement this approach by computing the confidence intervals as well, and ensuring that our static choice only chooses a non-default configuration when a sufficient confidence interval is achieved. Figure 3 demonstrates the importance having a significant number of samples to making meaningful decisions about choosing alternate configurations. In summary, the statistical approach predicts an optimal static configuration per query, and it predicts the corresponding performance increase from that configuration change. We refer to this approach in the results section as the *static optimal* choice.

9b) Machine Learning Model Training

In contrast, our machine learning approach predicts the optimal configuration not only as a function of the query itself, but also as a function of current system state. The intuition is that optimal JVM tuning is a function not only of code and hardware, but of current hardware state. This means that a system with no load may perform a different set of optimizations than a fully loaded system. Therefore, the goal of our predictor is then to take the current host state, and predict the optimal tuning configuration.

Recall that each row in the DynaJET database contains a data point comprising of three main components: host state, JVM configuration, and throughput metrics, all of which are encoded in a real-valued vector. The goal is to predict the optimal state, yet a sample only contains a measurement, not a label of the optimum. How do we create a 1:1 mapping such that each row corresponds to exactly one training example?

Instead of directly trying to determine the optimal configuration, we instead opt to train a regression model. The model takes the host state and JVM configuration as input, and predicts throughput. Conceptually, if the model predicts throughput accurately, we can use the model predict performance in different configurations, and select the configuration with the highest predicted throughput.

10) Model Deployment

Since the predictions must be done in real time, the machine learning model must be lightweight. Any time expended by the machine learning model must be recovered by its potential performance improvement. Therefore, we focus our efforts on lightweight regression algorithms from the SciKit learn library [20]. Even utilizing a brute-force approach that predicts the throughput of all considered configurations (up to 36), we find that all models require under 1/10th of a second of compute time, which is negligible compared to the minutes of run time for an average executor.

For each query, we try to fit the input data using linear regression, lasso regression, and random forest regression to predict throughput. To select the optimal model, we partition the data into a split of 80% train and 20% validation. Using k-fold cross-validation, we tune the model parameters of the each tested regressor. We then use the validation set to compare the different tuned models using R^2 score. R^2 score measures the model's explained variance against the variance in the data. A model that always predicts the same value regardless of input features would receive an R^2 score of 0, and a model that fully explains variance in the data would receive a score of 1.0. We then persist the model with the highest score on the validation set into the DynaJET controller package.

Once models have been trained and deployed as part of the package, standard DynaJET control flow ensues. First, the wrapper makes a decision about whether to choose an optimal configuration or a random configuration. In reinforcement learning, this referred to as an ϵ-greedy approach, where a exploration configuration is chosen with probability ϵ [21]. Otherwise, DynaJET attempts to optimize the JVM for throughput.

For throughput optimization, DynaJET will first look to see if a machine learning model is present for the current query. If not, DynaJET will defer to a predefined *static optimal* configuration. In the event a machine learning model is found, DynaJET utilizes the observed system state and tests potential configurations. For the scope of our work, we utilize a brute force approach since it incurs no notable penalty on overall performance. However, an alternate approach such as gradient ascent could be used to search the configuration space more efficiently if required in the future. We then select the configuration that was predicted to have the highest throughput and apply it to the JVM.

11) Model Updates

As mentioned, our framework supports continual updates naturally. Even after models are deployed, samples are continuously collected. Sub-optimal prediction and explicit exploration ensures continual learning about the space, but even optimal predictions continually report data points consisting of query, host state, JVM configuration and throughput. End users can choose to periodically perform steps 9a) and 9b) to ensure the tuning models remain up to date with a continual changing software stack, new queries, and new data.

4 Methodology

4.1 Data Collection

We detail our data collection process for collecting training data and JCT results. In a production environment, the approach would be to continually train and update models. However, to get the models started, an initial training step is required to accumulate enough data for statistically sound results. Thus, we conduct a separate training phase where we randomly permute the JVM parameters while continuously running our performance benchmark suite; queries overlap in execution, so they still generate load conditions that provide samples for different machine states at different executors.

We utilize an internal Spark performance benchmark suite maintained by an industrial data warehouse. The full benchmark suite contains over 700 unique queries, however, we find the the mass of compute in a much smaller subset of queries. Interestingly, this leads to better optimization for larger, more CPU time intensive queries. This is because larger queries create more executors providing more samples, while smaller, less important queries generate few executors and thus give us fewer samples. This is actually an ideal scenario, as it allows us to have tighter confidence intervals around important queries. As such, we only present results around important queries with sufficient samples for meaningful results. We utilize the top 10 most CPU intensive queries in the benchmark, which equate to approximately 25% of the CPU time for the full benchmark. We refer to these queries via letters A through J.

All results were collected on a 40-node test cluster at an industrial data warehouse. Each node is a two-socket, Intel Skylake system with a total of 40 cores and 80 threads. While the cluster is isolated from the production environment, it runs the full software stack that would be normally be in production, and by extension, is subject to the “datacenter tax” [22]. In addition to our work being a proof-of-concept, choosing a test cluster allows us to report more accurate data for JCT. Our test set of queries is submitted as batch, with no other jobs interacting unfairly skewing the results of a particular run or query. However, the queries do interact with each other, so dynamic resources, such as varying CPU and memory utilization, remain present.

4.2 JVM Tuning Configurations

We explore two modes JVM performance tuning: single parameter tuning and multi parameter tuning. The details of the configuration spaces explored are detailed in Table 1. As mentioned previously, the tuning parameters were chosen based on domain expertise that method inlining has significant impacts on performance in the data warehouse. We first explore the motivational parameter, `FreqInlineSize`. Our baseline configuration includes a non-default setting for this parameter, as `FreqInlineSize=128` was found to perform optimally for the data warehouse as a whole[1]. We further explore two other parameters, also related to JIT-compilation as described in section 2.1. Because we find no such cases in which increasing the `FreqInlineSize` parameter beyond 256 produces any positive results, we omit these configurations from the multi-parameter search space. While we do this manually, this approach is congruous to *beam search*.

Its worth noting that our configuration space is relatively small compared to the number of available parameters and settings. In order to report fair performance comparisons, we need to ensure that the entire software stack remains constant. Doing otherwise may result in us reporting performance results that are a function of unrelated layers, such as a changing datacenter tax. While we can postpone updates to an extent, we are still limited to a fixed amount of time. Keeping the search space relatively small enabled us to run all experimental configurations under the same conditions. In contrast, the gradual change of performance would be accounted for in a production deployment of DynaJET by continuously collecting samples, and periodically updating models.

[1]The Java default for `FreqInlineSize` is 325 B.

Single Parameter Tuning		
Parameter	Baseline	Explored Configuration
FreqInlineSize	128	64, 128, 192, 256, 325, 384, 448
Multi Parameter Tuning		
FreqInlineSize	128	64, 128, 192, 256
MaxInlineSize	35	25, 35, 50
ReservedCodeCacheSize	240m	192m, 240m, 284m

Table 1: JVM Parameter Test Configurations

5 Results

5.1 Job Completion Time

Our central results present four test configurations in comparison to the baseline described in the previous sections. The four configurations are as follows:

- *Baseline*: Fixed parameter setting found to be optimal across all queries in the performance benchmark suite.
- *Single Parameter Static*: tunes only the `FreqInlineSize` parameter using the statistical analysis described in 9a, setting the parameter at the granularity of per-query.
- *Single Parameter Dynamic*: uses machine learning regression models to predict the optimal configuration based on the current system state. In this experiment, we consider the same seven configurations as the static case, but we choose which configuration to apply dynamically.
- *Three Parameter Static*: same approach as the *Single Parameter Static*, but considers `FreqInlineSize`, `MaxInlineSize`, and `ReservedCodeCacheSize`.
- *Three Parameter Dynamic*: uses the same methodology as the *Single Parameter Dynamic* case, but considers three parameters. Predicts throughput for a total of 36 potential configurations given the current system state and selects the optimal.

Figure 4 presents the performance of each query relative to the baseline JVM tuning configuration. Each experiment was run 10 times to limit the amount of experimental noise caused by run-to-run variation which equates to results being within 1% of the mean with

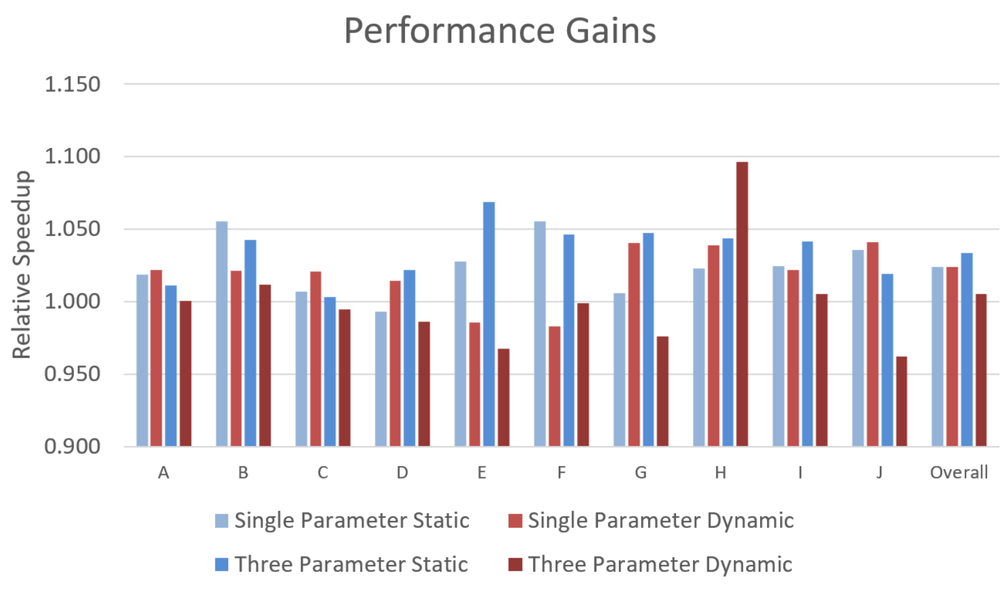


Figure 4: Central result of net performance gains across test suite of queries.

a 95% confidence interval. The X-axis represents queries that are labeled by their ID's in the test suite. The Y-axis represents JCT speedup, as measured by total CPU time.

For a single parameter, we see that the average change in performance is very similar for both the static per-query tuning and the dynamic tuning approach. Both methods net a performance gain of approximately 2.3% CPU time savings across our benchmark suite. While this number may seem low, a 2.3% performance gain relates to being able to remove an entire node in our 40-node cluster. At scale, this can translate to millions of dollars in datacenter CapEx and OpEx savings [22, 23].

The more interesting results, however, are revealed when looking at tuning three parameters. As expected, the static tuning approach provides much more consistent results, since it is based on statistically significant observations that are able to tightly bound confidence intervals. On average, the static per-query optimizations lead to a 3.3% performance gain over the global optimization techniques. Unfortunately, we find that the performance changes using the adaptive and predictive approach provides very inconsistent results. We see that query H, one of the most compute intensive queries in our test suite, experiences a 9.6% performance increase utilizing the dynamic tuning approach, compared to a 4.3% gain using a static tuning methodology. On the other hand, we see that in many cases, the dynamic tuning approach degrades performance by more than 3%.

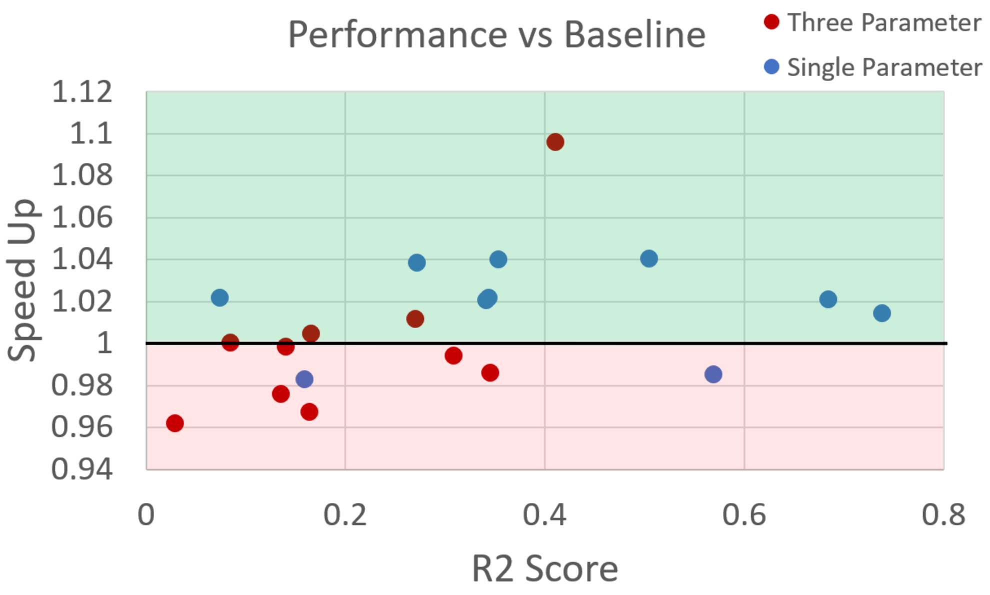


Figure 5: This plot examines the correlation between model performance (R^2 score) and query speedup. While a high R^2 score does not guarantee high performance, a low score is likely to negatively impact performance.

5.2 Debugging the Machine Learning Approach

Our first intuition behind why performance results are so inconsistent is to question the accuracy of the machine learning models themselves. As such, we perform an off-line experiment to test the accuracy of our machine learning models utilizing R^2 score. In Figure 5, we plot the R^2 scores of our machine learning models for each query against the query speedup.

At first glance, trends in the plot may not be evident. However, the average R^2 score for the single parameter tuning is in fact twice that for the multi-parameter scenario. In fact, only one multi-parameter model achieves a higher R^2 score than the average of all single parameter models. Moreover, the multi-parameter model that does achieve the highest R^2 score is also the model that achieves the highest performance improvement. Similarly, the multi-parameter model with the lowest R^2 score causes the most significant performance degradation. This leads us to conclude that while a higher R^2 score is not a guarantee for higher performance, deploying with a model with a low score certainly risks performance degradation.

We consider the scenario that a user may only choose to deploy models if their prediction

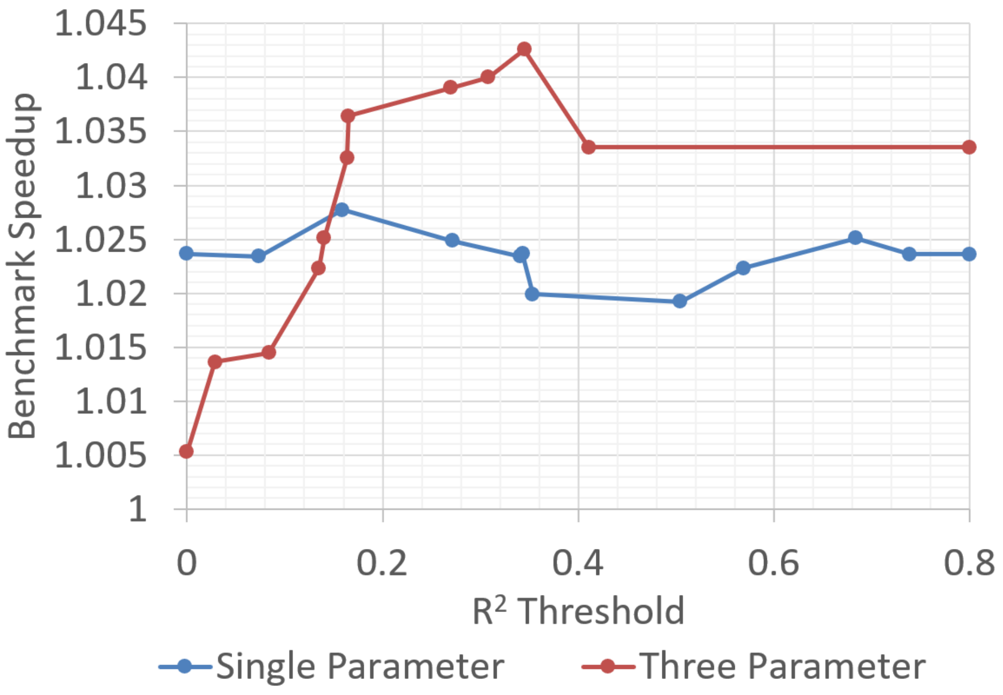


Figure 6: The graph above demonstrates how total speedup across all queries would change if a thresholding method was applied to choose whether or not to use a model for dynamic tuning. For example, if the threshold is set to a minimum R^2 score of 0.5 (corresponding to 0.5 on the x-axis), the three parameter tuning approach would default to using all static configurations, while the single parameter tuning approach would utilize a dynamic tuning approach for two of the ten queries.

quality meets a certain threshold. If the model's validation score is above the user defined threshold, it is deployed and JVM tuning parameters are configured dynamically. If the model's score is insufficient, we default back to the static per-query optimal tuning strategy. Using this methodology, we demonstrate in Figure 6 what overall performance would look like at various thresholds for both the single parameter and three parameter tuning scenarios. An R^2 threshold of 0 always chooses the dynamic model while an R^2 score of 0.8 always chooses the static approach. While setting a higher R^2 threshold ensures we do not risk degrading performance, both single and three parameter approaches demonstrate optimal performance using a balance of static and dynamic models. While it is unrealistic to assume we can perfectly choose the R^2 threshold in practice, it motivates that neither dynamic or static approach is ideal and instead a hybrid of the two approaches is always the best choice. Theoretically, a hybrid approach has the potential for an overall improvement of 4.2% in the three parameter case, and 2.5% in the single parameter case.

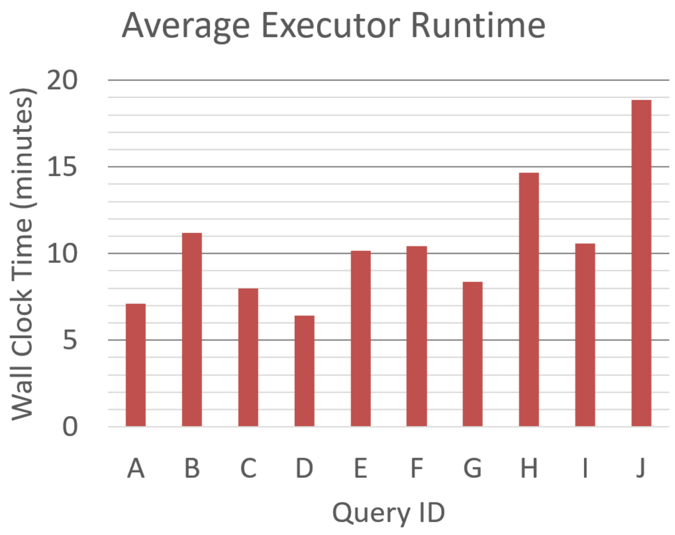


Figure 7: The average wall clock execution time of executor processes. While we capture an initial notion of system state, it is unlikely that the state remains consistent for the entirety of the executor, leading to poor predictions.

5.3 Diving Deeper

At this point, we have shown that while dynamic tuning has the potential to increase performance, poor model accuracy can limit its potential when considered against a static tuning approach. This raises the question: why is model accuracy low? We argue model accuracy suffers for two compounding reasons. Consider the set of input features our machine learning models use to predict optimal configuration: current CPU utilization, current memory utilization, and the average processes queued in the past 1, 5, and 10 minute windows. However, this notion of CPU and memory utilization is recorded from the 1/10th of a second before an executor is created, but, we find that executors run for 5-20 minutes of wall-clock time on average (Figure 7). This means our CPU and memory measurements are essentially using a measurement that corresponds to 0.027% of run time to predict the environment for the remaining 99.973%. Essentially, we have insufficient features and dynamic environment to accurately predict the future dynamic environment.

Despite this, we still assert that the ideal tuning configuration of the JVM really is a function of system environment. First, query H demonstrated a case where dynamic tuning of parameters improves JCT twice as much as that of a static configuration (9.6% vs 4.3%). This result is confirmed with a 95% confidence interval as well. Orthogonally, we provide attempted to train models to predict throughput using solely the tuning configuration, and no host state. For every query tested, utilizing only the configuration and omitting the host state caused our regression models achieve R^2 scores of effectively 0. This confirms that adding the host state explains some amount of performance variance, since the R^2 scores of our models which do incorporate host state are significantly more accurate than those that do not.

5.4 Discussion and Future Work

We have developed a proof-of-concept framework in an industry setting for JVM tuning for Spark. Using this infrastructure, we were able to answer three main questions about the future directions for extracting maximum performance from the Spark engine.

1) Does the specific query affect the optimal JVM tuning configuration, and if so, how much?
Our Java wrapper layer enables tuning JVM parameters specifically for the query being processed. Even in the scenario that Spark is simultaneously processing multiple queries (as is almost always the case), our infrastructure is able to pass sufficient metadata to tune accordingly for each executor. As a result, we find that tuning just three JVM parameters, statically per-query can improve performance up to 6.8%, and 3.3% on average in our benchmark suite.

2) Does JVM tuning depend on the current system state?
Prior to our work, automatic tuning methodologies focused on finding the best tuning configuration for a particular workload. However, our hypothesis is that not only does the tuning depend on the job, but also the active state of the host's environment. We find that our regression models are able to much more accurately predict performance when system state is taken into account, indicating that the JVM performance is a function of tuning configurations, workload, and system state.

3) Can we utilize machine learning to dynamically predict optimal JVM tuning configurations?
We have shown that machine learning approaches have the potential to boost performance significantly higher than static configurations. In six of the ten queries, our dynamic ML-based tuning approach outperforms a static approach. In the case of multiple parameters, we've shown that a dynamic approach can improve performance 5% more than a static approach. Yet, on average our machine learning approaches demonstrate inconsistent performance. To this end we motivate that future work should focus on utilizing better input features to more accurately predict future environment during the executor's lifespan.

6 Related Work

In recent history, black box tuning for software parameters has become increasingly prevalent. Rather than overhauling architectures, researchers are moving toward extracting performance by fine tuning knobs. Chen et al. utilize machine learning to tune Hadoop system parameters [11]. Jayasena et al. automate tuning the JVM by utilizing a tree structure to reduce the configuration space [10]. Bei et al. utilize random forests to tune Spark, but their approach requires end-to-end running of benchmarks [12]. Yaksha, proposed by Kamra et al., uses a control theory approach to tune performance in 3-tiered websites [24]. In the hardware auto-tuning domain, control theory tuning approaches are popular [25–27].

Several prior works are worth noting in more detail, as they provide a clear picture of the progression of the state-of-the-art. Released in 2009, iTuned is one of the first general approaches to SQL workload tuning [9]. The iTuned tool uses an adaptive sampling technique to efficiently search through large parameter spaces. It is one of the first works to suggest that tuning infrastructures should maintain their own internal database about performance statistics, and use that history to automate future experiments. Furthermore, to limit the impacts of experimentation, iTuned provides an automatic way to schedule additional experiments in a "garage" when nodes are under utilized.

OtterTune improves upon iTuned by applying state-of-the-art machine learning techniques [8]. Firstly, OtterTune uses dimensionality reduction techniques to identify important performance metrics. Next, OtterTune uses regularization to determine which performance tuning knobs are most important. OtterTune then begins its automated tuning process by considering the workload's overall behavior, and mapping it to a previously stored observed workload. OtterTune then utilizes Gaussian Process regression to efficiently search through the configuration space. To minimize variance, OtterTune further employs a technique called *intermediate knob selection*, which gradually adds more tuning parameters to the search space. This limits the variance of the process, leading to a quicker convergence to a near optimal configuration.

BestConfig is another similar approach for automatically tuning configurations [28]. Instead of using machine learning techniques in multiple phases to decompose the problem like OtterTune, BestConfig relies on a combined sampling and search algorithm. It utilizes divide and diverge sampling, which divides a high dimensional space into subspaces. It then combines the partitioning with a recursive bound and search technique to find a near-optimal point. BestConfig presents significant performance wins in a variety of cloud applications including Spark, Hadoop, Cassandra and more.

The previously noted approaches all utilize increasingly complex mathematical approaches to solve the problem of searching through a high-dimensional space. The previous works all assume that benchmarks must be run end-to-end to sample configurations, and expend great resources to make this sampling optimal. Rather than continue the trend of making incremental improvements to sampling and search algorithms, we try to solve the problem at an architectural level by making sample collection cheaper.

We develop an infrastructure that operates at the granularity of individual Spark executors. Since thousands of executors can be created in a single workload, we are able to achieve many more samples at much lower cost than prior approaches. Moreover, this same infrastructure allows us to make optimization decisions at much finer granularity, even accounting for the current system state to make the optimal decision. The trade-off is that in order to make fine-grain control practical, we are limited to much more simplistic models for both sampling and control of parameters. Still, it's worth noting that since our system architecture is decoupled into an online and offline layer, many of the previous approaches for input feature selection, parameter exploration, and search could be used in conjunction with our approach.

Finally, it is worth noting that few adaptive tuning strategies exist, however, these are mostly geared for very specific parameters rather than general approaches. Zhao et al. propose a dynamic tuning methodology for serialization strategy within Spark [29]. Tay and Tung develop a statistical approach to automatically tune database buffers [30].

7 Conclusion

In this work we presented DynaJET, an dynamic tuning framework for Spark. Collaboration with a large scale data warehouse exposed the importance of fine-grain parameter tuning, but the challenge of collecting sufficient, representative performance data to do so effectively. To solve this, DynaJET enables data collection and control of JVM parameters at the executor level. By isolating performance-related test configurations to individual executors, we are able to sample the search space at two to three orders of magnitude compared to prior approaches which require end-to-end experimentation of jobs. Moreover, our DynaJET accounts for the current system state at the host level to optimization for current conditions. Finally, because control occurs at the granularity of Spark executors, our framework lends itself to continual testing and sampling of alternate configurations. Models can be retrained at any point with the most up-to-date data from a production Spark cluster without having to perform offline test runs. We show that using just three JVM paremeters we can increase Spark JCT by 3.3% across our benchmark suite, and that dynamic control can achieve as high as 9.6% improvement for a single job.

References

[1] Apache, “Spark,” https://github.com/apache/spark.

[2] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin *et al.*, “Apache spark: a unified engine for big data processing,” *Communications of the ACM*, vol. 59, no. 11, pp. 56–65, 2016.

[3] DataBricks, “Spark+AI Summit Europe 2019,” https://databricks.com/sparkaisummit, 2019.

[4] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun, “Making sense of performance in data analytics frameworks,” in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, 2015, pp. 293–307.

[5] S. Agarwal, D. Liu, and R. Xin, “Apache spark as a compiler: Joining a billion rows per second on a laptop,” 2016.

[6] R. Xin and J. Rosen, “Project tungsten: Bringing apache spark closer to bare metal,” *Engineering Blog of Data bricks*, 2015.

[7] Oracle, “Java Hotspot VM Options,” https://www.oracle.com/technetwork/articles/java/vmoptions-jsp-140102.html.

[8] D. Van Aken, A. Pavlo, G. J. Gordon, and B. Zhang, “Automatic database management system tuning through large-scale machine learning,” in *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 2017, pp. 1009–1024.

[9] S. Duan, V. Thummala, and S. Babu, “Tuning database configuration parameters with ituned,” *Proceedings of the VLDB Endowment*, vol. 2, no. 1, pp. 1246–1257, 2009.

[10] S. Jayasena, M. Fernando, T. Rusira, C. Perera, and C. Philips, “Auto-tuning the java virtual machine,” in *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. IEEE, 2015, pp. 1261–1270.

[11] C.-O. Chen, Y.-Q. Zhuo, C.-C. Yeh, C.-M. Lin, and S.-W. Liao, “Machine learning-based configuration parameter tuning on hadoop system,” in *2015 IEEE International Congress on Big Data*. IEEE, 2015, pp. 386–392.

[12] Z. Bei, Z. Yu, N. Luo, C. Jiang, C. Xu, and S. Feng, “Configuring in-memory cluster computing using random forest,” *Future Generation Computer Systems*, vol. 79, pp. 1–15, 2018.

[13] M. Maas, T. Harris, K. Asanović, and J. Kubiatowicz, "Trash day: Coordinating garbage collection in distributed systems," in *15th Workshop on Hot Topics in Operating Systems (HotOS {XV})*, 2015.

[14] Y. Yarom, Q. Ge, F. Liu, R. B. Lee, and G. Heiser, "Mapping the intel last-level cache." *IACR Cryptology ePrint Archive*, vol. 2015, p. 905, 2015.

[15] P. Guide, "Intel® 64 and ia-32 architectures software developer's manual," *Volume 3B: System programming Guide, Part*, vol. 2, 2011.

[16] J. Laskowski, "The Internals of Apache Spark 2.4.2," https://jaceklaskowski.gitbooks.io/mastering-apache-spark/.

[17] A. Or, "Dynamic Allocation in Spark," https://www.slideshare.net/databricks/dynamic-allocation-in-spark, Databricks, 2015.

[18] A. C. De Melo, "The new linux'perf'tools," in *Slides from Linux Kongress*, vol. 18, 2010.

[19] A. Thusoo, Z. Shao, S. Anthony, D. Borthakur, N. Jain, J. Sen Sarma, R. Murthy, and H. Liu, "Data warehousing and analytics infrastructure at facebook," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010, pp. 1013–1020.

[20] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg *et al.*, "Scikit-learn: Machine learning in python," *Journal of machine learning research*, vol. 12, no. Oct, pp. 2825–2830, 2011.

[21] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[22] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks, "Profiling a warehouse-scale computer," *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3, pp. 158–169, 2016.

[23] D. C. Knowledge, "The Facebook Data Center FAQ," https://www.datacenterknowledge.com/data-center-faqs/facebook-data-center-faq, 2010.

[24] A. Kamra, V. Misra, and E. M. Nahum, "Yaksha: A self-tuning controller for managing the performance of 3-tiered web sites," in *Twelfth IEEE International Workshop on Quality of Service, 2004. IWQOS 2004*. IEEE, 2004, pp. 47–56.

[25] R. P. Pothukuchi, S. Y. Pothukuchi, P. Voulgaris, and J. Torrellas, "Yukta: multilayer resource controllers to maximize efficiency," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*. IEEE Press, 2018, pp. 505–518.

[26] S. Wang, C. Li, H. Hoffmann, S. Lu, W. Sentosa, and A. I. Kistijantoro, "Understanding and auto-adjusting performance-sensitive configurations," in *ACM SIGPLAN Notices*, vol. 53, no. 2. ACM, 2018, pp. 154–168.

[27] R. P. Pothukuchi, A. Ansari, P. Voulgaris, and J. Torrellas, "Using multiple input, multiple output formal control to maximize resource efficiency in architectures," in *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*. IEEE, 2016, pp. 658–670.

[28] Y. Zhu, J. Liu, M. Guo, Y. Bao, W. Ma, Z. Liu, K. Song, and Y. Yang, "Bestconfig: tapping the performance potential of systems via automatic configuration tuning," in *Proceedings of the 2017 Symposium on Cloud Computing*. ACM, 2017, pp. 338–350.

[29] Y. Zhao, F. Hu, and H. Chen, "An adaptive tuning strategy on spark based on inmemory computation characteristics," in *2016 18th International Conference on Advanced Communication Technology (ICACT)*. IEEE, 2016, pp. 484–488.

[30] D. N. Tran, P. C. Huynh, Y. C. Tay, and A. K. Tung, "A new approach to dynamic self-tuning of database buffers," *ACM Transactions on Storage (TOS)*, vol. 4, no. 1, p. 3, 2008.