INTROSPECTIVE COMPUTING

by

Karl Taht

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

Department of Computer Science

The University of Utah

May 2020

The University of Utah Graduate School

STATEMENT OF DISSERTATION APPROVAL

The dissertation of **Karl Taht**

has been approved by the following supervisory committee members:

Rajeev Balasubramonian,	Chair(s)	__ Date Approved
Ryan Stutsman,	Member	__ Date Approved
Vivek Srikumar,	Member	__ Date Approved
Mary Hall,	Member	__ Date Approved
Chris Wilkerson,	Member	__ Date Approved
Ganesh,	Member	__ Date Approved

by **Ross T. Whitaker**, Chair/Dean of

the Department/College/School of **Mathematics**

and by **David B. Keida**, Dean of The Graduate School.

ABSTRACT

Todo.

For my parents, Georg and Claire.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

NOTATION AND SYMBOLS

α	fine-structure (dimensionless) constant, approximately $1/137$
α	radiation of doubly-ionized helium ions, He++
β	radiation of electrons
γ	radiation of very high frequency, beyond that of X rays
γ	Euler's constant, approximately $0.577\,215\,\ldots$
δ	stepsize in numerical integration
$\delta(x)$	Dirac's famous function
ϵ	a tiny number, usually in the context of a limit to zero
$\zeta(x)$	the famous Riemann zeta function
$\ldots$	$\ldots$
$\psi(x)$	logarithmic derivative of the gamma function
ω	frequency

ACKNOWLEDGEMENTS

The people who surround us remind us of who we are. The influences of my friends, family, and my wife Lyndsey have undoubtedly made this journey possible.

CHAPTER 1

INTRODUCTION

1.1 The Advent of Cloud Computing

The new millennium has brought a paradigm shift computers. No longer do developers maintain bulky workstation PCs or small businesses maintain a server room. Instead, we all login to the ethereal ”cloud” of seemingly endless compute to fulfill our computing needs.

1.1.1 Thesis Statement

As traditional performance scaling slows, we expect industry response to manifest in the form of increasingly complex hardware and software which expose and demand the end users to set performance related parameters. We hypothesize that intelligent, automated frameworks will be imperative in tuning performant and efficient computing systems.

1.2 Characterizing Program Behavior

This thesis is organized into three main sections, utilizing a bottom-up approach. We begin by developing a novel methodology for characterizing program phase behavior in real-time. We build upon these finding by then developing a dynamic tuning framework which responds based on system and application behavior. Finally, we

1.3 Exploiting Program Behavior

, we develop a methodology for characterizing program phases in real-time. We find that using key hardware performance counters, programs can be classified into discrete program phases.

1.3.1 The POP Detector

1.3.2 DynaJET

1.3.3 Intorspective Computing

1.3.4 Organization of this Dissertation

CHAPTER 2

BACKGROUND

CHAPTER 3

THE POP DETECTOR: LIGHTWEIGHT, ONLINE PROGRAM PHASE DETECTION

3.1 Introduction

Time-varying behavior of a program promotes the need for systems to adapt to changing requirements. Thus, architects constantly seek to develop more adaptive systems which can better respond to specific applications and conditions. At the lowest level, hardware features such as branch predictors and data prefetchers constantly monitor changing instruction and data streams to update their heuristic predictors. These online learning agents employ lightweight algorithms that can not only predict within a matter of cycles, but also update within a matter of cycles.

In particular, the update rules for these agents must be chosen through careful analysis across many workloads. One analysis technique is to generate an S-Curve, a graph which encapsulates the effects of changing a feature across many workloads. The goal is to maximize the area under the curve: the average performance across workloads. In Figure 3.1, we utilize the S-Curve to show the effects of disabling the L2 spatial prefetcher on an Intel® system. Because the area under the curve is close to 1, we can see that most of the workloads examined are agnostic to the spatial prefetcher. However, there are a few outlier workloads in which performance could be improved. In this simple example, a natural solution would be to disable the prefetcher for these workloads.

However, we additionally plot an S-Curve for the individual program phases of *lbm_s* from Spec2017 [36] in Figure 3.2. From a global perspective, *lbm_s* demonstrates a marginal performance improvement with the spatial prefetcher enabled. However, on a per-phase basis we see that this same program has negative outliers as well. This motivates the need to adapt during particular intervals, or phases, of a program.

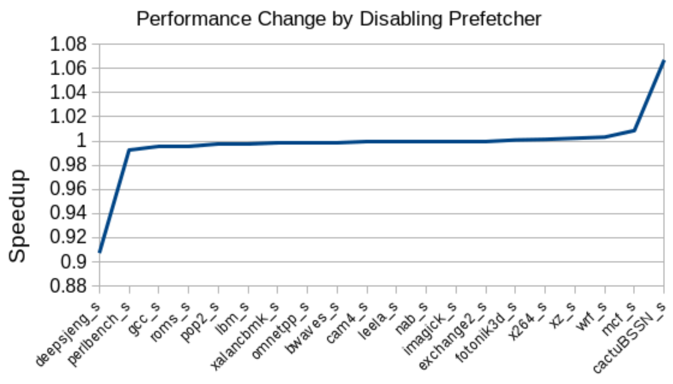


Figure 3.1. Spec2017 performance change by disabling spatial L2 prefetcher, using training dataset.

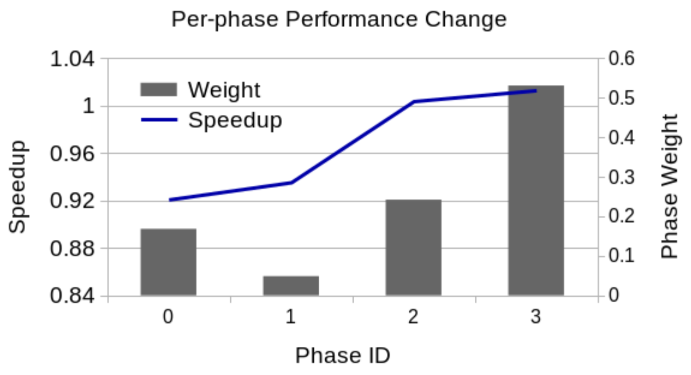


Figure 3.2. The lbm_s performance change (denoted by the S-Curve) per phase via disabling spatial L2 prefetcher, with respective phase weights denoted by bars. The phase weight refers to the amount of time the program spends in a particular phase, relative to the program as whole. As a result, the sum of phase weights is always 1.

Much work has been done to address this need for dynamic optimization, ranging from careful offline phase analysis techniques, to lightweight online phase detectors, and more recently control theory [4, 5, 33, 34, 40]. Each of these approaches has merit. Basic block vectors (BBVs), for example, have proved to be an excellent baseline that has stood the test of time for phase detection. Yet, this algorithm requires a plethora of data making it infeasible for online algorithms. As such, multiple prior works have addressed approximating BBVs, originally through theoretical hardware support and more recently on a real-system using Precise Event-Based Sampling (PEBS) [4, 5]. Ultimately, the goal remains to understand individual phases and optimize features on a per phase basis, which can be defined as a set of requirements.

Functionally, any phase detector must produce phases of similar, repeating behavior. However, using program phases for online performance optimization necessitates a number of requirements for a successful phase detection algorithm. Firstly, the phase detector must operate independently of performance changes. For a phase detector to enable analysis (and learning) of different hardware configurations, it must produce phases which are of similar program execution so performance can be compared. Yet this comparison can only be made if the phases detected remain *invariant* to performance knobs (i.e., phase A is still labeled phase A even after a change in a performance knob). Since both SimPoint and ScarPhase are based off code execution, they both satisfy this requirement.

From a feasibility perspective, a phase detection technique should be feasible in commodity hardware with minimal program and system perturbation. Collecting BBVs, the input for SimPoint, is infeasible in real-time. As such, detectors like ScarPhase employ sampling to mitigate this overhead. However, we find that even at large 100M instruction intervals, the overhead of such a technique is more than 3%, and if finer granularity is required, the overhead increases exponentially. This overhead is undesirable in the scope of online performance optimization.

In this work, we present the POP detector which drastically reduces phase detection overheads to as little as 0.09%, while performing competitively with existing state-of-the-art phase detection performance. Specifically, we outline our contributions as follows:

- We propose a new phase detector metric, Statistical Time Analyzing Baseline (STAB) which captures the trade-offs of phase interval length, phase stability, and the number of phases.

- We introduce the Precise Online Phase (POP) detector, a real-time phase detection algorithm which leverages performance counters for accurate, fine-grain phase detection with significantly lower overhead than existing approaches.

- We perform a detailed competitive analysis between the POP detector and the state-of-the-art ScarPhase detector, using existing metrics as well as our proposed metric.

We organize our paper as follows. In section 3.2, we discuss the different phase detection algorithms discussed throughout the paper, including the POP detector. The key distinguishing factor of the POP detector is the specific performance counters selected, which we detail in section 3.3. In section 3.4, we describe the metrics for evaluating phase detection algorithms including our new metric, STAB. We then perform a case study regarding phase interval size in section 3.5, before performing extensive analysis between ScarPhase and the POP detector in section 3.6. Finally, we provide a brief survey of related work in section 3.7 before concluding in section 3.8.

3.2 Phase Detection Algorithms

We briefly describe the phase detection algorithms used throughout the paper, including our proposed POP detection algorithm.

3.2.1 SimPoint: Offline Baseline

SimPoint traces code execution in order to classify a program into phases [19]. The original work stems from the notion of a basic block: a region of code with exactly one entry and one exit. A Basic Block Vector (BBV) is generated by profiling the number of instructions executed within each basic block for a period of time. SimPoint performs approximate K-Means clustering on the BBVs, and utilizes the Bayesian Information Criterion (BIC) to select the optimal number of phases for a program. It then outputs a number of BBVs which serve as simulation points. We treat each simulation point as a cluster center, which allows us to label each interval of execution according to the closest simulation point.

3.2.2 ScarPhase: Online Baseline

ScarPhase samples conditional branch instruction pointers (IPs) to detect program phases [5]. The framework is built utilizing the Linux *perf* subsystem and operates as a user-space program. Intel®'s Precise Event-Based Sampling (PEBS) enables sampling IPs directly at a granularity of up to once per 25K instructions. The branch IP's are hashed into a vector to construct a signature for each interval. The signature is fed to an online leader-follower clustering algorithm to identify program phases. In order to mitigate the overhead, ScarPhase makes use of a buffer in which the kernel writes directly that allows processing only at interval granularity. Additionally, ScarPhase employs a dynamic sampling frequency methodology which reduces the sample rate during stable phases of execution. As the dynamic sample rate is itself predictive, it slightly degrades phase detection accuracy in favor of lower overhead. All data reported relative to ScarPhase uses the authors' original code, available on GitHub [6].

3.2.3 The POP Detector

The POP detector was motivated by the non-trivial overhead associated with ScarPhase, as well as its performance degradation when using a phase length of less than 100M instructions. To address this, the POP detector utilizes only performance counters to detect program phases. The rationale for using performance counters is two fold. Firstly, interrupts are required only at the end of interval to collect performance counts. This drastically reduces overhead, as we will show. Secondly, the counts are true measurements rather than samples. This means that when performing increasingly fine-grained phase detection, the underlying data does not incur additional loss.

The quality of phase detection using performance counters relies on selecting specific counters which capture code execution paths rather than performance related metrics. If the counters are related to performance, they create a feedback loop. Consider the prefetcher example, where the goal is to choose whether to enable or disable the feature. If one of the counters used to generate the signature was the number of cache misses, disabling the prefetcher would likely affect the cache miss count. As a result, the signature would change, and potentially trigger a false phase change. Therefore, the performance counters must relate specifically back to code execution paths in order to be agnostic to

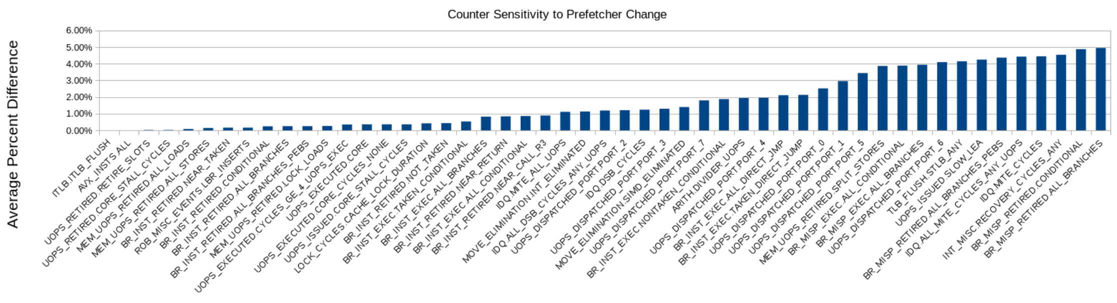


Figure 3.3. Interval-to-Interval average percentage difference from varying prefetchers. Data collected using Spec2006 benchmarks.

system changes. We dedicate section 3.3 to our novel approach which targets optimal counter selection via machine learning.

Using this key set of performance counters, the POP detector periodically measures counter values. Similar to ScarPhase, this is done by leveraging the Linux *perf* subsystem in a user-space program. The counter values are treated as the interval's signature, and fed as input into an online leader-follower clustering algorithm. This lightweight online algorithm takes just 10,000 cycles, or about 5μs at 2GHz, to cluster a new point. The leader-follower algorithm automatically generates new clusters when the similarity between the newest data point and existing cluster centers is too large. We measure distance using average percentage difference, and set the similarity threshold to 20%. We allow the POP detector to track up to 32 unique phases simultaneously. If a new cluster is detected after 32 unique phases have been detected, we employ an LRU replacement policy to remove a cluster.

3.3 Performance Counter Selection

In this section we describe our process for selecting key performance counters which are able to capture program phases which relate back to repeating segments of code, rather than repeating behavior. Our two-step approach examines over 200 candidate counters, and ultimately provides a subset of just 8 through the use of machine learning techniques and a ground truth of SimPoint phases.

3.3.1 Performance Counter Collection

Other than limiting our search to per-core events, we offer no domain expertise to try to reduce the set of performance counters. This results in a list of over two hundred performance counters to profile for our Haswell-based Intel® system. Because the hardware offers only 8 events to be simultaneously collected, the most common approach is to group events and use time-multiplexing to approximate counts. However, because the goal of our study is to *avoid* sampling error, we instead collect over 50 separate traces, each with real counter values to eliminate the possibility of sampling error.

3.3.2 Ensuring Counter Invariance

Our first pass is to remove performance counters which correlate to system behavior rather than program execution. At first glance this may not seem intuitive, as similar system behavior implies similar code execution. While this is true, it is best to think of the program's instructions as the input, and the system's behavior as a result. A good phase detector correlates intervals which have similar *inputs*. This enables analysis of the output, including performance, in different system configurations. In other words, we seek to eliminate any performance counters which relate to performance.

To find a set of performance counters which relates back to code execution rather than system behavior, we perform a test to measure change as a result of system reconfiguration. Our system has four hardware prefetchers which can be enabled or disabled through a Model Specific Register (MSR) [1]. The default configuration, prefetch MSR $= 0$, enables all prefetchers, while the disabled configuration, prefetch MSR $= 15$, disables all prefetchers. We measure the average percentage difference of each performance counter on a per interval basis. For a run of N intervals, we measure the difference by comparing the counts on interval i in prefetcher configurations 0 ($c_{i,0}$) and 15 ($c_{i,15}$):

$$\text{Avg \% Diff} = \frac{1}{N}\sum_{i=1}^{N}\frac{|C_{i,0} - C_{i,15}|}{(1/2)(C_{i,0} + C_{i,15})}$$

Figure 3.3 shows a ranking of the top 50 performance counters which responded least to changes in prefetcher configuration, making them potential candidates. Many of the low variance counters relate back to instruction mix, as expected. In order for a program counter to advance to our next stage, we require less than 1% average percentage difference due to prefetcher reconfiguration.

3.3.3 Counter Pruning

Our final step ranks program counters by their ability to accurately predict phases based on code execution. We collect Basic Block Vectors (BBVs) and use SimPoint to determine ground truth program phases. A naive approach would be to train a series of predictors using every possible combination of eight counters, and select the counters with the best prediction accuracy. Unfortunately, this is infeasible from a compute perspective as choosing eight counters from just the 20 best candidates involves testing over 125,000 combinations. Moreover, such a method provides no insight as to a ranking of the counters.

Thus, we employ a decision forest classifier to predict the SimPoint phases. After training the decision forest, we can view the importance associated with each variable, measured using the Gini impurity metric [30]. Using this approach across Spec2006 benchmarks [35], we rank the variables as shown in Table 3.1.

Lastly, we validate that the counters do in fact capture data similar to that of basic block vectors by performing a distance visualization experiment. Motivated by the original work by Sherwood et al. [39], we measure similarity between signatures at different intervals to find repeating patterns. A darker color indicates greater similarity, with black being an exact match. Similarly, white is a complete mismatch. The results for *mcf* workload are shown in Figure 3.5. Note that this figure captures only signature similarity over the course of program execution, and include no notion of program phase.

By examining the *mcf* visualization, it is obvious that performance counters capture the same pattern as BBVs. However, while BBVs can indicate complete mismatch, performance counters always retain some element of similarity (indicated by the gray in place of the white). As a result, the phases given by the performance counter signatures are more generalized than BBV phases. While the difference may seem significant, we will later show in the results section that the difference only results in slightly worse stability compared to IP-based approaches.

Counter	**Normalized Score**
UOPS_RETIRED.RETIRE_SLOTS	100
UOPS_RETIRED.CORE_STALL_CYCLES	93.28
BR_INST_RETIRED.ALL_BRANCHES	79.29
BR_INST_RETIRED.NOT_TAKEN	77.16
MEM_UOPS_RETIRED.ALL_STORES	74.58
BR_INST_RETIRED.NEAR_TAKEN	69.17
MEM_UOPS_RETIRED.ALL_LOADS	64.75
UOPS_EXECUTED.CYCLES_GE_4_UOPS_EXEC	64.60

Table 3.1. The highest ranked counters using the Gini impurity metric.

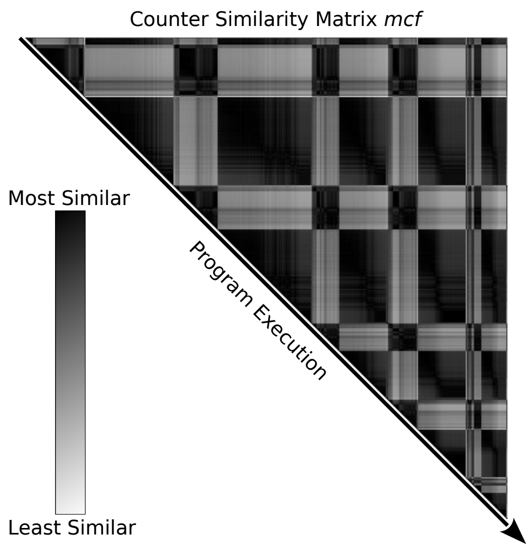


Figure 3.4. Similarity matrix for *mcf* using performance counters.

3.4 Phase Detection Evaluation

In this section we present a number of metrics used for evaluating phase detection algorithms. We utilize CPI as the basis to compute all metrics. For example, μ_i refers to the standard deviation of CPI among all intervals labeled as i. Note that all computations presented are done post-execution as a means to compare program phase detectors. Many of such metrics could be adapted to be computed online, but to remain consistent with prior art and fair competitive analysis, we utilize these metrics a posteriori.

3.4.1 Phase Stability

A method for measuring the quality of a phase detection algorithm is to look at the stability (a percentage). Stability is a statistically grounded metric, which measures the *Coefficient of Variation* (CoV) within each phase. CoV is simply a normalized method to compute variance, more formally,

$$\text{CoV} = \frac{\sigma}{\mu}$$

The stability metric also factors duration of the program spent in each phase, or the phase's *weight*. For N phases, the stability can then be computed as:

$$\text{Stability} = 1 - \sum_{i=1}^{N} \text{CoV}_i * \text{Weight}_i$$

However, a naive phase detection algorithm could classify each interval as a unique phase, pushing the stability to 100%. To correct this, Sembrant et al. [5] proposed the *Corrected Coefficient of Variation* (CCoV). CCoV penalizes algorithms for detecting more unique phases by adjusting the basic CoV computation. CCoV places all phases which are dissimilar to their neighbors as part of a virtual phase. All samples in the virtual phase are given the CoV of the program as a whole. This gives the desirable property that in both trivial cases, a single phase or all unique phases, the CCoV will converge to the same value.

3.4.2 Phase Interval Length

The previous metric may be affected by phase interval length, but do not use it as a weighting metric to favor smaller or larger intervals. As such, we propose a new metric: *Statistical Time Analyzing Baseline* (STAB). This metric examines the balance between number of phases, stability, *and* phase interval length. STAB is a metric with the application of phase detection algorithms for real-time optimization in mind.

In order to dynamically tune a system, performance must be assessed in a given configuration. Consider an algorithm which must choose between the default configuration A and a new configuration B. In order for the algorithm to move to configuration B, the performance difference must exceed measurement error for the change to be reasonable. Thus, we may need to sample both configuration A and B multiple times to establish a small enough margin of error with enough confidence.

For clarity, we describe a specific example. Consider that a program has $3\times$ more stability at 100M instruction intervals versus 10M instruction intervals. There exists a quadratic relationship between stability and the number of samples required to establish a given confidence interval and a particular error tolerance [21]. Given that, we can compute that at 100M instruction intervals, x^2 samples are required, whereas at 10M instruction intervals, $(3x)^2$ samples are required. This means for the 100M instruction intervals, we would spend $100x^2$ instructions establishing a baseline, but for 10M instruction intervals, we would only spend $90x^2$ instructions. As we show in the results sections, this leads to some non-linear trade-offs with shorter phases.

We begin the formulation of STAB by computing the number of samples needed for a given confidence interval and error tolerance by the formula:

$$\text{Num Samples} = \left(\frac{Z * CoV}{Error}\right)^2$$

where Z is a factor related to the confidence interval (e.g. 95% confidence interval Z-value is 1.96) [21]. STAB calculates the number of samples required for a given confidence requirement and error tolerance on per-phase basis. These samples are then summed for each phase seen to give the total number of samples required to establish a performance baseline. Thus for a program with N phases:

$$\text{Baseline Samples} = \sum_{i=1}^{N}\left(\frac{Z * CoV_i}{Error}\right)^2$$

Next, we account for phase interval size, to determine the total number of instructions the program would need to spend establishing this baseline. This is simply baseline samples multiplied by the phase interval size. Finally, the Statistical Time spent Analyzing a Baseline (STAB) can be computed as a ratio:

$$\text{STAB} = \frac{\text{Baseline Samples} * \text{Interval Size}}{\text{Total Instructions}}$$

The result of this computation is an intuitive ratio: how many instructions (as a percentage) would be spent establishing a confidence interval with a given error margin. However, this ratio is affected by the total duration of the program, an undesirable artifact. For instance, Spec2017 *omnetpp* has just one phase, and requires 50 samples to establish baseline performance using either the training or reference dataset[1]. However, because the training set is smaller and has fewer instructions than the reference set, the unnormalized version of STAB is 7.3% for the training dataset, but just 0.91% for the reference dataset. Thus, we choose to normalize by a fixed number of instructions so that the total run length of the program does not affect STAB.

$$\text{STAB}_{norm} = \frac{\text{Interval Size} * \text{Baseline Samples}}{\text{100B Instructions}}$$

Using this formulation, *omnetpp* has a STAB_{norm} of 5% with either the training or reference datasets. For the remainder of the paper, we utilize the normalized version of STAB with 100B instructions as the normalization factor. This choice is arbitrary and has no bearing on the final results as it only linearly scales the values. We report normalized STAB for a given confidence interval c and error tolerance e as $\text{STAB}_{c,e}$. For example a confidence interval of 95% and error tolerance of 5% would be $\text{STAB}_{95,5}$.

3.5 Case Study: Interval Size

Utilizing our STAB metric, we perform an analysis to understand the trade-offs of phase detection at various interval sizes. To create a fair baseline, which does not lose information or incur overhead, we perform trace-based analysis using SimPoint, as described in section 3.2.1. We utilize QEMU[20], a full-system emulator, to collect accurate, complete traces of long-running programs. Once an instruction trace is collected, we can generate BBVs and corresponding phase labels for any desired interval size.

[1]This measurement was performed using ScarPhase with 100M instruction intervals and dynamic sampling enabled.

To perform analysis with CoV and STAB we require performance data, specifically the IPC for each interval. We collect 50 performance traces per workload and average the per interval data to mitigate run-to-run error. Using 19 Spec2006 workloads with the training dataset, we perform the SimPoint analysis at instruction intervals ranging from to 1M to 128M. The results are shown in Figure 3.6.

Firstly, note that both CoV and STAB are lower-is-better metrics. As expected, the smaller intervals have an increasingly high CoV. However, this is non-linear in respect to the number of intervals available for analysis. While COV increases from 4.5% to over 15%, the number of intervals available increases by 128×. As a result, the $\text{STAB}_{95,5}$ metric decreases from 10% at 128M instruction intervals, to just 1.9% at 1M instruction intervals.

While the STAB metric conveys the benefits of utilizing smaller granularity phases, it neglects to consider overhead. We define overhead as the additional time required for the program to execute with a phase detection framework versus without any monitoring framework. We analyze the overhead of both ScarPhase and the POP phase detector at 1M, 10M, and 100M instruction interval sizes utilizing wall-clock time and plot the results in Figure 3.7. At 100M instruction intervals, we measured ScarPhase to incur 3.19% overhead on average across Spec2017 speed benchmarks, whereas the POP detector incurs just 0.09% overhead. At 10M instruction intervals this overhead rises to an undesirable 16.12% for ScarPhase, but just 1.35% for the POP detector. While the STAB metric continued to improve at smaller intervals, even the more efficient POP detector causes a 9.65% increase in execution time at 1M instruction intervals. Because of the goal of the POP detector is low-overhead, we do not present 1M instruction interval results in the following section.

3.6 Results

3.6.1 Methodology

To assess the proposed POP detector, we compare it to the prior art, ScarPhase, utilizing the Spec2017 benchmark suite on an Intel® system[2]. Both phase detectors are configured with a similarity threshold of 20% for their clustering algorithm, the default specification for ScarPhase. To obtain consistent, repeatable results we evaluate single-threaded

[2]Performance results are based on testing as of February 13, 2019 and may not reflect all publicly available security updates.

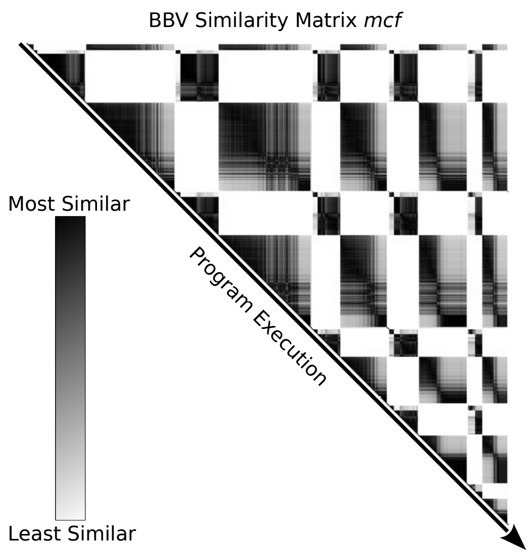


Figure 3.5. Similarity matrix for *mcf* using Basic Block Vectors

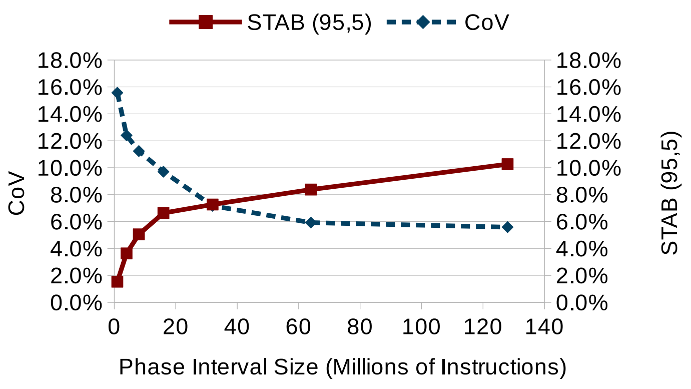


Figure 3.6. Trade off between CoV and STAB using various interval sizes. While CoV suffers at smaller intervals, many more intervals are available for analysis. As a result, the STAB overhead reduces at smaller intervals.

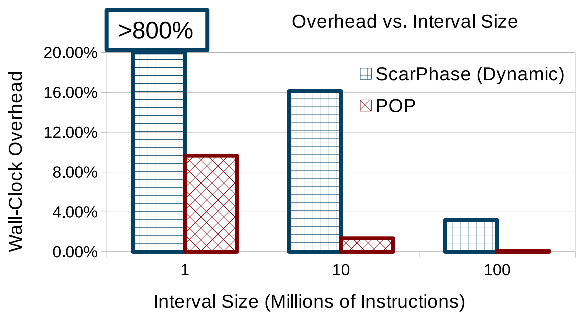


Figure 3.7. Wall-clock Overhead comparison. Results averaged across Spec2017 benchmarks using the training dataset.

applications via *speed* benchmarks. It is also known that single threaded applications typically have more phase variability, making them more applicable to our analysis [4]. Furthermore, we apply a few command line boot parameters to ensure that results are not skewed. Table 3.2 details each of our Intel® Haswell evaluation node's specifications.

Note that both the POP detector and ScarPhase are implemented as user-space programs which tap into the Linux *perf* subsystem, allowing them to collect counter and branch IP information. This means that no super-user privileges are required to run either detector. While kernel-module implementations can offer superior performance, the user space implementation allows us to perform a fair comparison to the prior art. As Sembrant et al. [5] mention, context switches to this user space implementation can incur significant overhead at smaller time scales but is often more desirable for system administrators.

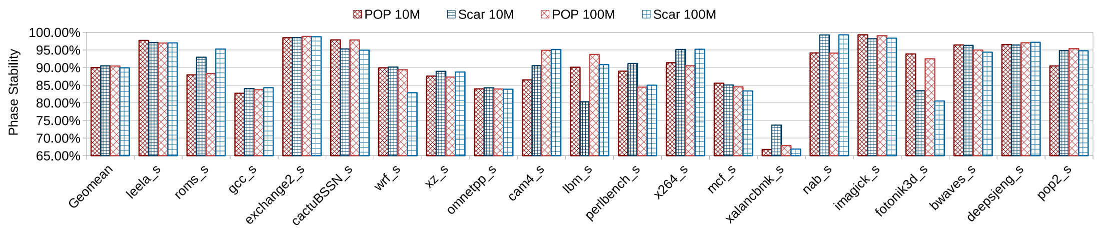


Figure 3.8. Phase Stability metric for Spec2017 Speed benchmarks, Reference Dataset.

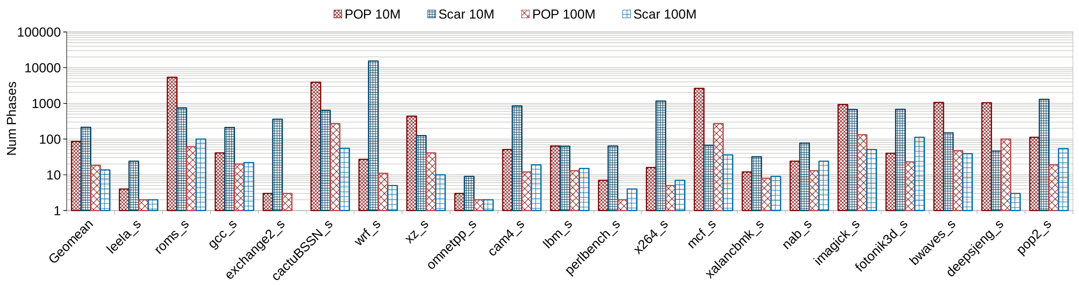


Figure 3.9. Number of Phases detected for Spec2017 Speed benchmarks, Reference Dataset.

3.6.2 Phase Detector Performance

We perform an in-depth comparison of ScarPhase and the POP detector. As discussed in Section 3.2.2, ScarPhase has an optional dynamic sampling mode which reduces over-

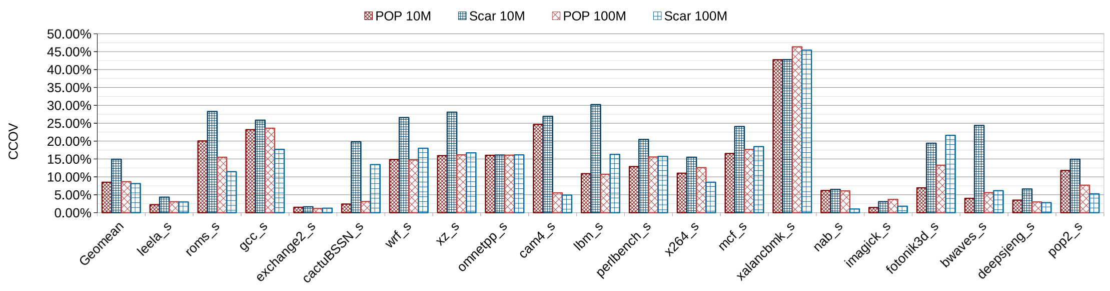


Figure 3.10. Corrected Coefficient of Variation (CCoV) metric for Spec2017 Speed benchmarks, Reference Dataset.

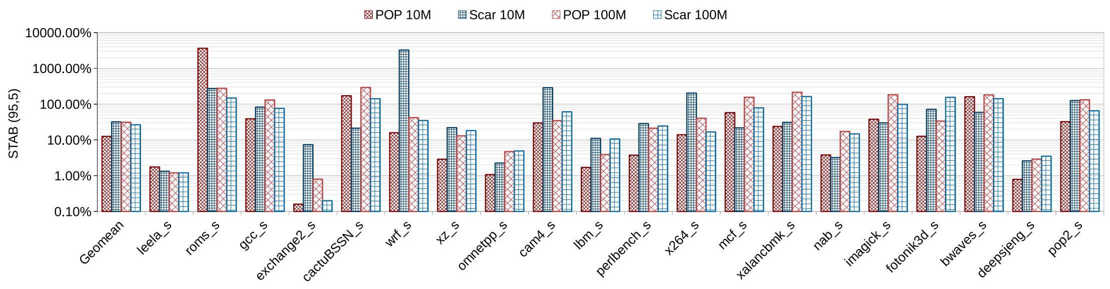


Figure 3.11. Normalized $STAB_{95,5}$ metric for Spec2017 Speed benchmarks, Reference Dataset.

head while slightly degrading phase detection performance. To remain succinct and capture ScarPhase's optimal phase detection ability, we focus our discussion of results on ScarPhase without dynamic sampling. We report results for both 100M and 10M instruction intervals, the smallest granularity possible without significant overhead. We include all *speed* benchmarks in the Spec2017 benchmark suite, which includes some outliers. As such, we utilize the geometric mean to draw final conclusions for various metrics. Note that when we use the term "average", we are referring to the geometric mean unless otherwise stated.

3.6.2.1

Stability We begin by examining phase stability in Figure 3.8. The POP detector finds 0.51% more stable phases at 100M instruction intervals. In particular, POP performs significantly better on lbm and fotonik3d, where its stability is more than 10% higher than ScarPhase at 100M instruction intervals. These programs are among the most variable with respect to IPC changes in the Spec2017 benchmark suite, but ScarPhase inadequately separates the phases. In the case of fotonik3d at 100M instruction intervals, ScarPhase generates 112 unique phases, but more than 85% of intervals reside in just 3 phases. As a result, the phases are poorly separated and phase stability is just 80.5%.

At 10M instruction intervals, ScarPhase is able to identify phases which are 0.52% more stable compared to the POP detector. Nevertheless, for half of the benchmarks tested the POP detector is within a 2% margin compared to ScarPhase, and never performs more than 7% worse than ScarPhase. Also worth emphasizing is that this result is with ScarPhase's dynamic sampling mode disabled. Enabling dynamic sampling mode nets a 2% decrease in average stability for ScarPhase, placing it behind the POP detector in terms of phase stability.

3.6.2.2 Number of Phases

While the two detectors perform similarly from a phase stability standpoint, they begin to diverge when analyzing the number of phases detected, as shown in Figure 3.9. At 100M instruction intervals, ScarPhase detects 13.7 phases compared to 18.5 with the POP detector. Yet the reverse is true at 10M instruction intervals, where ScarPhase detects 214 unique phases compared to POP's 86, a reduction of nearly 60%. Specifically, for 12 of the 20 benchmarks tested, ScarPhase finds more than 100 unique phases. While more unique phases at finer granularity is intuitive, the diverged of ScarPhase and the POP detector can be traced back fundamental difference of sampling versus measurement. ScarPhase's signature vector is more susceptible to noise in the case of finer granularity due to its lack of sufficient samples. This result is particularly prominent in x264 and wrf, where the number of phases detected increases by more than order of magnitude.

3.6.2.3 CCoV

Sembrant et al. proposed the "Corrected" Coefficient of Variation as a means to account for the intrinsic trade-off between number of phases and stability. We compute the CCoV and report the results in Figure 3.10. The CCoV metric most emphasizes the effectiveness of the POP detector at smaller phase intervals. ScarPhase's CCoV degrades by 83% at finer granularity due to the large number of phases detected. The lower-is-better metric increases from 8.12% to 14.93% when moving from 100M to 10M instruction intervals using

ScarPhase. By comparison, the POP detector experiences negligible deviation in CCoV, improving from 8.65% to 8.49% when moving from 100M to 10M instruction intervals.

3.6.2.4 STAB

While CCoV balances number of phases and stability, the STAB metric applies statistics to also balance phase interval size. We report the normalized STAB metric using a 95% confidence interval and 5% error tolerance in Figure 3.11. The overall winner here is the POP phase detector with 10M instruction intervals. ScarPhase's optimal configuration using 100M instruction windows boasts a $STAB_{95,5}$ of 26.5%, 2.1× worse than the POP detector's 12.5% $STAB_{95,5}$ with 10M instruction intervals. Based on our statically grounded metric, this means use of the POP detection framework would net significantly quicker analysis to enable better online optimization.

3.6.3 Summary of Results

We provide a summary of results in Table 3.3 to highlight the key takeaways of the POP detector compared to the existing state-of-the-art, ScarPhase. All values reflect the geomean across all benchmarks tested; the same data presented in Figures 3.7,3.8,3.9,3.10, and 3.11. While the two phase detection frameworks are closely matched in the traditional phase detection metrics, the two parameters that apply most directly to the success of deploying an online phase detection algorithm are execution overhead and STAB. For any potential performance gain, the optimization will have to overcome the overhead of the phase detection framework. Our experiments concluded that even in the most optimistic scenario with dynamic sampling enabled and larger 100M instruction intervals, ScarPhase still incurs a 3.19% overhead. POP detection incurs just 1.35% execution overhead at shorter 10M instruction intervals, and just 0.09% with 100M instruction intervals.

In addition to overhead, an adaptive system must adequately analyze performance before applying a final configuration. Motivated by this, we propose the STAB metric to statically quantify the amount of time an adaptive system must spend analyzing behavior to gain a particular level of confidence with a given error tolerance. This metric captures trade-offs of phase stability, number of phases, *and* phase interval size. STAB quantifies that POP detection enables 2.1× faster performance analysis compared to ScarPhase.

3.7 Related Work

Sherwood et al. began to define the standards for program phase detection in their early work which utilized basic blocks. Their initial work did an in-depth dive to understanding the cyclical behavior of a program via frequency analysis [39]. They later expanded this work to SimPoint, which attempts to cluster basic vectors to define phases [38]. While random projections and approximate K-means clustering helped to speed up SimPoint, the fundamental idea still hinged on collecting basic blocks which are expensive to profile. Dhodapkar and Smith attempted to mitigate this overhead by proposing a hardware mechanism which constructs signatures hashing branches into a n-bit vector [3]. Sherwood et al. later refine this proposal by adding in the amount of time spent in each branch as well as a phase predictor [40]. Dhodapkar and Smith then provided an evaluation framework utilizing sensitivity, false-positives, and stability which lay the foundation and motivation for CCoV and STAB [17]. In their comparison work, they find that while BBVs provide the best phase detection, utilizing branch counts nets a phase detection with 80% of the performance. In many ways, this is the motivation for our work.

Shen et al. explore the notion of phases as repeating, but non-uniform behavior [45]. They utilize wavelet filtering and allow for variable phase sizes. While their technique performs comparably to manual-code injection, it requires a separate training step to determine program phases. Nagpurkar et al. also approach the problem utilizing dynamic phase sizes by suggesting the notion of stable and transition periods [31]. They use an adaptive trailing window policy to detect phases and instrument a new binary. However, their technique is only applicable to Java applications. Nevertheless, both of these approaches are particularly applicable when the goal is optimizing software as the program can be run and profiled many times. Since these publications, Linux *perf* has improved dramatically, and supports such analysis. Gregg has an excellent blog which includes many tutorials for such software optimization using *perf* and other tools [10].

However, another application of phase detection is online hardware optimization, which requires lightweight online phase detection. In this scope, ScarPhase [5] represents the state-of-the-art. ScarPhase bridges the gap of previous phase detection techniques by designing a framework which works on present hardware, operates completely online without prior training, and has significantly lower overhead than previous approaches [5].

However, ScarPhase chooses 100M instruction window phases as a baseline by citing prior work. We differ from ScarPhase in that we explore trade-offs associated with different phase sizes. We show that while variance increases with finer resolution, the trade-off is nonlinear and has auxiliary benefits.

3.8 Conclusion

In this paper we explored program phase detection for the use case of online performance optimization. To perform real-time optimization, an algorithm must be able to evaluate per-phase performance in a particular configuration. The faster per-phase performance can be evaluated with a phase detection algorithm, the quicker a reinforcement learning agent can adapt the system. With this in mind, we create a new metric, STAB, which weights phase stability, number of phases, and interval size. It statistically grounds time required for a given algorithm to establish per-phase performance.

Utilizing STAB, we perform an oracle study using SimPoint and show that smaller phases are desirable to minimize learning overhead. However, we find that the existing state-of-the-art phase detector, ScarPhase, suffers from excessive run-time overheads when attempting fine-grain phase classification. In addition to overhead, ScarPhase's sampling techniques break down due to limited information, resulting in hundreds of unique phases being identified. To attempt to fill this void, we explore the use of performance counters for online phase detection.

We employ statistical and machine learning techniques in a two-step process to select a core subset of performance counters. Using these counters, we build the POP phase detector, which is able to accurately detect phases at fine granularity while incurring just 1.35% overhead. In the context of online optimization, the POP detector requires $2.1\times$ fewer instructions to establish baseline performance. If even lower overhead is required, POP requires just 0.09% overhead when using 100M instruction interval phases.

Base Board	Intel® Server Board S2600WT2
CPU	2x Intel® Xeon® CPU E5-2699 v3 @ 2.30GHz
BIOS	SE5C610.86B.01.01.0027.071020182329
DRAM	24x 16G DDR4 ECC DIMM @ 1600 MHz
OS	CentOS 7.5 w/ Linux 4.14.58
Kernel Cmdline	nmi_watchdog=0 transparent_hugepages=never cpuidle.off=1 intel_idle.max_state=0 cpufreq.off=1 intel_pstate=disable procesor.max_state=0 isolcpus=0,1 rcu_nocbs=0,1 rcu_nocb_poll nohz_full=0,1 skew_tick=1
Hyperthreading	Off

Table 3.2. Evaluation System

Geomean	**Overhead**	**Stability**	**Phases**	**CCoV**	**STAB**
POP 10M	1.35%	89.99%	85.8	8.49%	12.53%
Scar-D 10M	16.12%	88.39%	195.2	16.72%	53.30%
Scar 10M	17.85%	90.51%	213.9	14.93%	31.98%
POP 100M	0.09%	90.45%	18.5	8.49%	31.18%
Scar-D 100M	3.19%	88.62%	12.7	9.57%	35.04%
Scar 100M	9.01%	89.94%	13.7	8.12%	26.53%

Table 3.3. Summary of results for phase detectors at both 10M and 100M instruction window sizes. ScarPhase with dynamic sampling is listed as *Scar-D*, while with fixed sampling as just *Scar*. Overhead is reported in wall-clock execution time, and the *STAB* metric refers to 100B instruction normalized *STAB* with a confidence interval of 95% and error tolerance of 5%, as described in 3.4.2.

CHAPTER 4

DYNAJET: DYNAMIC JAVA EFFICIENCY TUNING

4.1 Introduction

The lines have begun to blur between high performance computing and big data analytics frameworks, as the scope and scale of these two areas have grown [43]. From the big data side, Spark has emerged as one of the most popular frameworks thanks to its ease of use, generality, and scalability [9]. Motivated by this, recent work has begun to explore leveraging Spark specifically for HPC applications [14, **?**]. Yet, the flexibility of analytics frameworks can also be their Achilles's heel: optimal performance is highly dependent on tuning configuration [12, **?**]. This is a particularly difficult problem since the compute stack often exceeds hundreds of tuning knobs, and often interact with each other creating a non-linear optimization space.

As such, significant research presents strategies to tune database systems and compute engines [44, **?**, **?**, **?**, **?**]. The problem is typically framed as follows: given a set of tuning knobs, design an automatic process that optimizes performance. Prior work has answered key questions including how to determine the most important tuning knobs and how to select meaningful performance metrics. Extensive algorithms also detail efficient methods for searching through the configuration space to quickly converge to a near optimal solution [44, **?**, **?**]. In this way, parameter tuning is essentially reduced to a multi-variable optimization problem.

Optimization algorithms span a wide range, with different algorithms having different trade-offs between algorithmic complexity and statistical bounds on the search space. For instance if data collection is cheap, favoring $\mathcal{O}(1)$ approach such as random sampling could be a good choice. On the other end of the spectrum, when collecting data is expensive it's worth leveraging compute to efficiently search the space. The general practice of collecting performance information requires running workloads end-to-end in test configurations to collect a <Configuration, Performance> data point. Since this sampling is fundamentally expensive, it's not uncommon to see algorithms which require $\mathcal{O}(n^3)$ complexity, with respect to the number of data points. OtterTune uses a Gaussian process approach requiring just that [44].

Besides the scale of most compute clusters, there additional reasons why test configuration performance data is limited. To ensure that end users do not experience unwanted performance degradation and SLO violations, trials are normally isolated from the production environment. These trials often consist of a benchmark suite which runs workloads that are representative of the production environment on separate test infrastructure. Alternative approaches also exist, such as iTuned's suggestion to schedule workloads during under under-utilization of the compute cluster [18]. Yet, these intervals must be sufficiently long to collect meaningful data and threaten to perturb users if utilization increases during testing phases.

While running end-to-end tests is required for some parameters, the approach is not without drawbacks. First, it can be very difficult build a representative benchmark suite, particularly as production workloads change over time. Second, even if the benchmark is representative, optimization is generic and applies a fixed configuration to all production workloads. Third, even in scenarios where production workloads are tested end-to-end during under utilized intervals they risk compromising data quality. This is because under utilized systems may prefer very different configurations than when resources are fully contended. For instance, a machine with high resource contention may prefer a more conservative tuning approach reducing memory and cache footprint[2]. In contrast, more aggressive settings may improve job completion times in an under utilized system, since additional resource use would have no penalty. Finally, all of these approaches require investing additional compute hours to collect performance statistics.

In this work, we shift away from incremental algorithmic improvements and instead focus on building a tuning framework which makes data collection both inexpensive to collect and representative of the production environment. We ensure that data is representative of production by enabling tuning and measurement on production workloads themselves, yet mitigate negative impacts of sub-optimal test configurations via fine-grain control and sampling. As a result, we are able to produce several orders of magnitude more data points compared to end-to-end measurements. In addition, because control is available at runtime, we can optimize for not only the current job being processed, but also respond to the current system environment (e.g. CPU and memory utilization).

Our proposed Dynamic Java Efficiency Tuner (DynaJET) framework is built around the Apache Spark compute engine [47]. Spark has a unique layered architecture where workers, called Spark Executors, are created and destroyed during a job's lifetime. Each worker is encapsulated in its own process, and more specifically its own Java Virtual Machine (JVM). In this way, Spark's architecture lends itself to fine grain tuning. DynaJET supports measurement, analysis, and configuration of JVM parameters at the Spark executor level.

Our framework comprises of two central components: a real-time "Java Wrapper" that measures and adapts the JVM parameters, and an offline component that trains and analyzes predictive models. The online component is accomplished via an additional layer in the software stack between Spark Container Managers and Spark Executors. This layer intercepts the creation of a new JVM and uses both query information *and* the current system state to predict an optimal configuration. The prediction mechanism is trained by a separate offline approach. As each instance of the Java wrapper runs, it records a new sample to a persistent database. This database is used to train machine learning models that enable optimal configuration prediction. The trained models are deployed to the real-time layer via a software package. Models can be retrained with new data as frequently as desired, and released via software packages updates.

The remainder of the paper is organized as follows. In section 4.2 we motivate why JVM tuning for Spark is important, and briefly cover the Spark architecture. Next, we provide a detailed description of DynaJET's architecture in section 4.3. In section 4.4 we review the methodology of our approach and discuss various comparison points. We present the dynamic tuning results in section 4.5, including additional analysis of our machine learning approach. We provide a brief survey of related work in section 4.6 before concluding in section 4.7.

4.2 Background

4.2.1 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 [28]. 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 **??**.

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 [46]. Moreover, because L1 and L2 caches are effectively shared because of hyper-threading [22], interactions between threads change effective size. This is why we explore tuning these parameters dynamically, based on system load, and by extension, cache pressure.

4.2.2 Spark Architecture

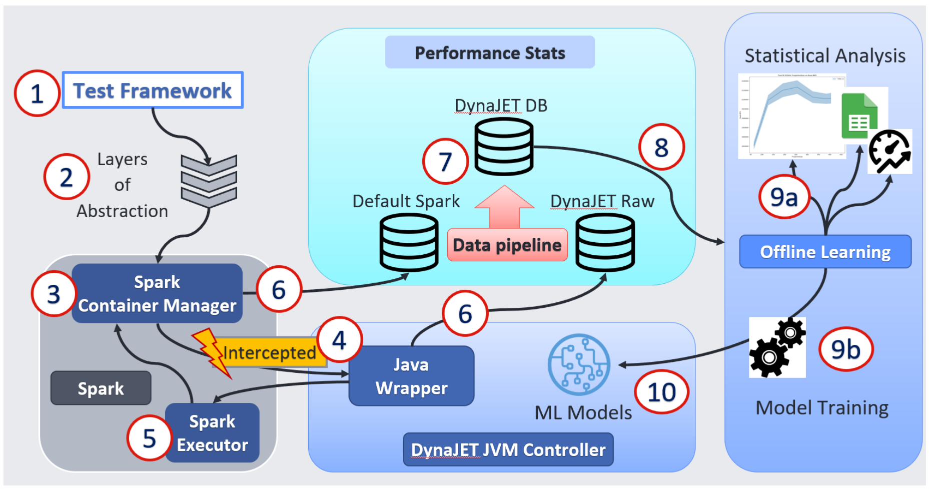


Figure 4.1. Graphical overview of the DynaJET Architecture. The details of each component and dataflow are described in detail in Section 4.3.

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 [27]. 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 [29]. 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, cluster managers 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.

4.3 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 4.1, labeled in the order in which we discuss components of the framework.

4.3.1 The JET Dataflow

4.3.1.1 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 to the JVM tuning controller. In a similar fashion to metadata, we pass "flags" to the JVM tuning controller via environment variables.

4.3.1.2 Layers of Abstraction

As noted in Section 4.2.2, the Spark architecture comprises of various levels of abstraction. The metadata encoded in the environment propagates through various layers of the compute stack.

4.3.1.3 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.

4.3.1.4 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 CPU and memory load, captured as a percentage. We also use historical load via the number of queued processes in the past 1, 5, and 15 minute marks. A more detailed description can be found in Table 4.1. 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 4.3.1.8.

4.3.1.5 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 [16]. This enables us to collect hardware statistics at the executor level, such as instruction and cycle counts, as well as cache and branch misses.

4.3.1.6 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*.

4.3.1.7 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 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

Feature	Description
CPU Utilization	System wide CPU utilization, measured for 100ms before executor execution begins.
Memory Utilization	Percentage of virtual memory allocated reported by Linux.
1-Minute Load Average	Number of processes in the system run queue, averaged over the last 1 minute
5-Minute Load Average	Number of processes in the system run queue, averaged over the last 5 minute
15-Minute Load Average	Number of processes in the system run queue, averaged over the last 15 minute

Table 4.1. System feature vector fields used for predicting optimal configuration.

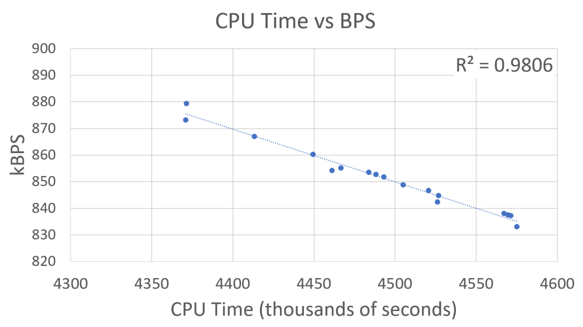


Figure 4.2. Scatter plot of 17 different runs of the same query plotting average throughput in kilobytes per second (kBPS) against JCT (CPU time). The nearly perfect linear relationship between BPS and CPU time validates that optimizing executors for BPS should optimize overall run time.

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 4.2, 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*.

4.3.1.8 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 [41]. 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 4.3 demonstrates the importance of 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 [32]. 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 [37]. 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.

4.4 Methodology

4.4.1 Data Collection

We build a custom Spark performance benchmark using test queries maintained by an industrial data warehouse. The data warehouse maintains over 700 unique queries to validate both performance and correctness of their Spark compute engine. In the interest of performance evaluation, we focus on the most time consuming test queries, and refer to these queries via letters A through J. These queries a wide range of CPU intensity, ranging from sub 1M bytes per second to over 20M bytes per second (per executor). Each query in our test suite processes at least 500GB, and some over 2TB, which equates to between 180 and 550 CPU hours per query.

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. The cluster runs the full software stack that would be normally be in production, and by extension, is subject to the “datacenter tax” [25]. Each run involves submitting queries as a batch, mimicking a production-like scenario due to random scheduling decisions which cause job interaction resulting in variable CPU and memory utilization during the duration of the job’s makespan. To ensure results are accurate, we re-run experiments until all 95% confidence intervals are within 1% (at least 10 times). In total, our results capture several years of CPU time statistics. All query speedup refers to the CPU time usage of each query.

4.4.2 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 Tables 4.2 and 4.3. 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 4.2.1. 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.

Parameter	**Baseline**	**Explored Configuration**
FreqInlineSize	128	64, 128, 192, 256, 325, 384, 448

Table 4.2. JVM Single Parameter Test Configurations

Parameter	**Baseline**	**Explored Configuration**
FreqInlineSize	128	64, 128, 192, 256
MaxInlineSize	35	25, 35, 50
ReservedCodeCacheSize	240m	192m, 240m, 284m

Table 4.3. JVM Multi Parameter Test Configurations

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

4.5 Results

4.5.1 Data Collection Rate

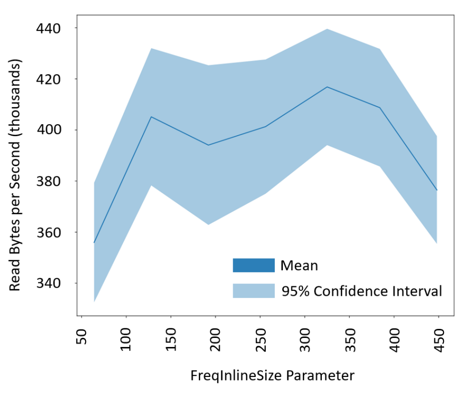
As motivated in our introduction, the one of the primary benefits of shifting to a dynamic, executor-level methodology is the ability to sample at much finer granularity. Prior end-to-end testing would require specifying a tune configuration and running a query from end-to-end. For instance, running the benchmark suite of queries our 40-node cluster takes about 1 hour of real-world time for a single configuration. In comparison, executor level metrics are collected at rate of over 4300 samples per hour. This is because most queries generate between 250-700 executors for a single run.

Additionally, this fine grain control means that test configurations no longer require separate test queries at all. This is because our executor-level infrastructure allows for administrators to specify a sampling ratio that ensures acceptable performance degradation to ensure SLO at met. This trade-off is visualized in figure 4.4, where lowering the sampling rate mitigates performance degradation. In comparison, running a test configuration from end-to-end can have significantly more adverse affects. This is because some exceptionally poor configurations degrade performance more than 10% in many cases. Executor level control averages out these negative outliers such that random configurations within our search space net only a 2% degradation from in total CPU time.

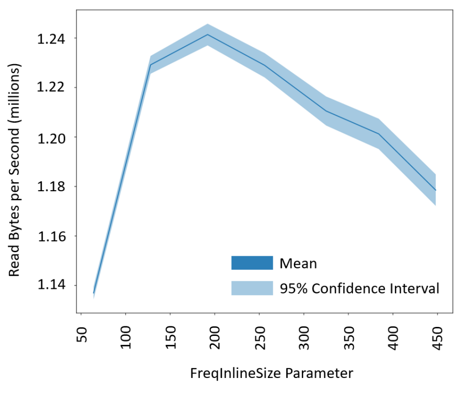
4.5.2 Performance Results

Performance results reflect the reduction in CPU time 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.



(a) Query with only 320 samples



(b) Query with over 6000 samples

Figure 4.3. Both graphs plot the mean performance in seven different `FreqInlineSize` parameter values, with the confidence intervals plotted in light blue. The left graph illustrates the case of 45 samples per configuration and fails to tightly bound expected performance. By comparison, the query on the right demonstrates the effects of having 20× the number of samples, resulting in much tighter confidence intervals.

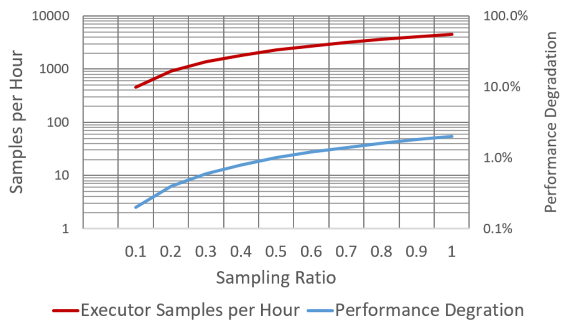


Figure 4.4. Sampling and performance implications of random configuration testing. DynaJET's executor level obfuscates the sampling cost to just 2% at most, because we never have to run the worst configuration from end-to-end. DynaJET also allows for sampling to be reduced, if further performance degradation mitigation is desired.

- *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 4.5 presents the performance of each query relative to the baseline JVM tuning configuration.The X-axis represents queries that are labeled by their ID's in the test suite. The Y-axis represents reduction in 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.

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%.

4.5.3 Understanding Dynamic Tuning Challenges

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 4.6, 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,

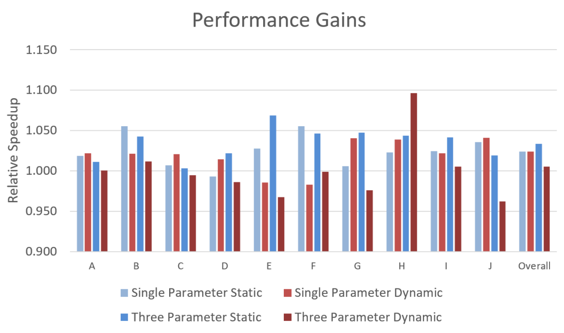


Figure 4.5. Central result of net performance gains across test suite of queries.

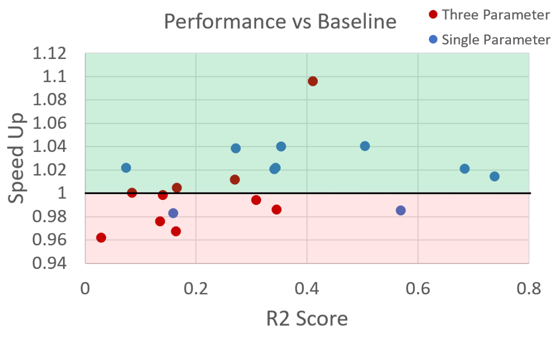


Figure 4.6. This plot examines the correlation between model performance (R^2 score) and query speedup. While a high R^2 score does not guarantee high performance, a low score is likely to negatively impact performance.

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 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 4.7 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.

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? While executors are dynamically created and destroyed, the JVM parameters must be selected at the time of the executors creating. As such DynaJET measures the current state of the machine, and uses this information to make a decision about the JVM parameters for the executor. However, we find that executors run for 5-20 minutes of wall-clock time on average, as shown in Figure 4.8. Since system state is dynamic, the creation time measurements may not reflect the future environment in the following 5-20 minutes of execution.

Despite this, we still assert that the ideal tuning configuration of the JVM really is a

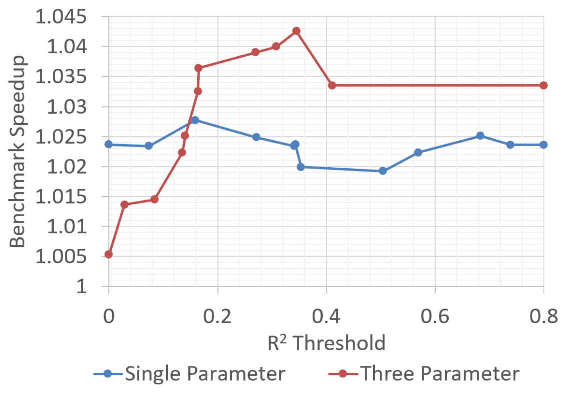


Figure 4.7. The graph above demonstrates how total speedup across all queries would change if a thresholding method was applied to choose whether or not to use a model for dynamic tuning. For example, if the threshold is set to a minimum R^2 score of 0.5 (corresponding to 0.5 on the x-axis), the three 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.

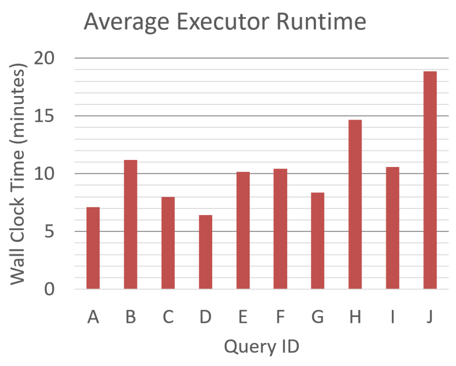


Figure 4.8. The average wall clock execution time of executor processes. While we capture an initial notion of system state, it is unlikely that the state remains consistent for the entirety of the executor, leading to poor predictions.

function of system environment based on two key insights. Firstly, we tested a predictors which utilize only configuration parameters and no host state to predict throughput. All regression models R^2 scores dropped below 0.1, inidicating that system state input certainly has an effect on throughput. Second, dynamic prediction did lead to some significant performance improvements. In the case of query H, CPU time savings improved twice as much as that of a static configuration (9.6% vs 4.3%), confirmed with a 95% confidence interval. Nevertheless, future work could include an additional predictor to estimate *future* system state for the executors projected lifetime. In theory, accurate prediction of the future environment that the executor will actually run in could be more useful than measurement at time of creation.

4.5.4 Interpreting Performance at Scale

At scale, small performance improvements can translate to millions of dollars in datacenter capital expenditures (CapEx) and operation expenditures (OpEx) savings [26, **?**]. For instance, on a 1000 node Spark Cluster using current Amazon EMR pricing, every 1% performance improvement translates to approximately $300,000 saved per year [8]. From the perspective of energy consumption, 1% improvement translates to 65MWh saved per year, assuming a 750 watt average consumption per server. Moreover, 65MWh savings translates to 32 tons of CO2 emissions reductions annually, based on 2016 data [13] [2].

4.5.5 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,

[2]Note that many industrial data warehouses utilize renewable energy resources with much lower emissions than the U.S. national average [7, **?**]

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.

4.6 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 [15]. Jayasena et al. automate tuning the JVM by utilizing a tree structure to reduce the configuration space [23]. Bei et al. utilize random forests to tune Spark, but their approach requires end-to-end running of benchmarks [11]. 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 [33, **?**, 34].

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 [18]. 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 [44]. 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 [49]. 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 [48]. Tay and Tung develop a statistical approach to automatically tune database buffers [42].

4.7 Conclusion

In this work we presented DynaJET, a 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.

CHAPTER 5

INTROSPECTIVE SCHEDULING

A modern data warehouse consists of a plethora of scheduling decisions, ranging from job scheduling across compute clusters to choosing the next latency-critical packet to process. Each scheduling algorithm is subject to a different set of design requirements, such as throughput requirements, compute budget, and integrating varying levels of fairness and prioritization. Yet all schedulers must balance two high level objectives: utilizing resources efficiency while maintaining quality of service requirements.

Significant work has gone into the two extreme ends of scheduling. Throughput-oriented batch compute tasks require efficient use of compute resources. As such, cluster management schedulers target high utilization of machines through efficient bin-packing by analyzing task resource requirements and machine availability (TETRIS, Borg)[]. In contrast, cloud microservices and packet processing require latency optimizations. As such, scheduling algorithms in this domain focus on optimization of low-level operations such minimizing context-switch overhead (SHINJUKU, DUNE) [].

Cluster schedulers typically focus on hard resource constraints. They attempt to match tasks with machines which have enough physical CPUs, memory, or network bandwidth for the incoming task. A valid first order requirement, especially when the inbound number of tasks can be exceed 10,000 tasks per minute (BORG)[]. Yet within a single node, performance isolation of tasks is not guaranteed. While modern Linux features such as cgroups help to mitigate some interference, low-level resources such as memory bandwidth and cache pollution.

To help with this, modern processors support explicit resource isolation techniques, such as memory bandwidth and last level cache partitioning. Unfortunately, this partitioning typically reduces overall throughput,

work asks the question if rather than targeting performance isolation, we can instead

APPENDIX A

THE FIRST

This is an appendix. Notice that the LaTeX markup for an appendix is, surprisingly, `\chapter`. The `\appendix` command does not produce a heading; instead, it just changes the numbering style from numeric to alphabetic, and it changes the heading prefix from **CHAPTER** to **APPENDIX**.

Blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah. Blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah. Blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah. Blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah. Blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah. Blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah. Blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah. Blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah. Blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah.

APPENDIX B

THE SECOND

This is an appendix.

Blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah. Blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah. Blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah. Blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah. Blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah. Blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah. Blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah. Blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah. Blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah. Blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah.

APPENDIX C

THE THIRD

This is an appendix.

There are several books [?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?] listed in our bibliography.

We also reference several journal articles [?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?] and three famous doctoral theses of later winners [?, ?, ?] of the Nobel Prize in Physics (1922, 1933, and 1921):

Notice that, even though those citations appeared in LaTeX `\cite{...}` commands with their BibTeX citation labels in reverse alphabetical order, thanks to the `citesort` package, their reference-list numbers have been sorted in numerically ascending order, and then range-reduced.

Mention should also be made of a famous Dutch computer scientist's first publication [?].

Font metrics are an important, albeit low-level, aspect of typesetting. See the *Adobe Systems* manual about that company's procedures [?].

The bibliography at the end of this thesis contains several examples of documents with non-English titles, and their BibTeX entries provide title translations following the practice recommended by the American Mathematical Society and SIAM. Here is a sample entry that shows how to do so:

```
@PhdThesis{Einstein:1905:NBM,
  author =       "Albert Einstein",
  title =        "{Eine Neue Bestimmung der Molek{\"u}ldimensionen}.
                 ({German}) [{A} new determination of molecular
                 dimensions]",
  type =         "Inaugural dissertation",
  school =       "Bern Wyss.",
  address =      "Bern, Switzerland",
  year =         "1905",
  bibdate =      "Fri Dec 17 10:46:57 2004",
  bibsource =     "http://www.math.utah.edu/pub/tex/bib/einstein.bib",
  note =          "Published in \cite{Einstein:1906:NBM}.",
  acknowledgement = ack-nhfb,
  language =      "German",
  advisor =       "Alfred Kleiner (24 April 1849--3 July 1916)",
  URL =           "http://en.wikipedia.org/wiki/Alfred_Kleiner",
  remark =        "Received August 19, 1905 and published February 8,
                  1906.",
  Schilpp-number = "6",
}
```

The `note` field in that entry refers to another bibliography entry that need not have been directly cited in the document text. Such cross-references are common in BibTeX files, especially for journal articles where there may be later comments and corrigenda that should be mentioned. Embedded `\cite{}` commands ensure that those possibly-important other entries are always included in the reference list when the entry is cited. The last bibliography entry [**?**] in this thesis has a long `note` field that tells more about what some may view as the most important paper in mathematics in the last century.

When entries cite other entries that cite other entries that cite other entries that . . . , multiple passes of LaTeX and BibTeX are needed to ensure consistency. That is another reason why document compilation should be guided by a `Makefile` or a batch script, rather than expecting the user to remember just how many passes are needed.

BibTeX entries are *extensible*, in that arbitrary key/value pairs may be present that are not necessarily recognized by any bibliography style files. The `advisor`, `acknowledgement`, `bibdate`, `bibsource`, `language`, `remark`, and `Schilpp-number` fields are examples, and may be used by other software that processes BibTeX entries, or by humans who read the entries. `DOI` and `URL` fields are currently recognized by only a few styles, but that situation will likely change as publishers demand that such important information be included in reference lists.

In BibTeX `title` fields, braces protect words, such as proper nouns and acronyms, that cannot be downcased if the selected bibliography style would otherwise do so. In German, all nouns are capitalized, and the simple way to ensure their protection is to brace the entire German text in the title, as we did in the entry above.

The world's first significant computer program may have been that written in 1842 by Lady Augusta Ada Lovelace (1815–1852) for the computation of Bernoulli numbers [**?**, **?**]. She was the assistant to Charles Babbage (1791–1871), and they are the world's first computer programmers. The programming language *Ada* is named after her, and is defined in the ANSI/MIL-STD-1815A Standard; its number commemorates the year of her birth.

We do not discuss mathematical *transforms* in this dissertation, but you can find that phrase in the index (except that this sample thesis doesn't have one!)

Blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah. Blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah. Blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah.

Blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah. Blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah. Blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah. Blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah. Blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah. Blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah blah.

REFERENCES

[1] *Disclosure of H/W prefetcher control on some Intel processors*. Accessed: 2018-10-12.

[2] *Intel Resource Director Technology (Intel RDT)*. Accessed: 2019-4-12.

[3] A. S. Dhodapkar and J. E. Smith, *Managing multi-configuration hardware via dynamic working set analysis*, in Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on, IEEE, 2002, pp. 233–244.

[4] A. Sembrant, D. Black-Schaffer, and E. Hagersten, *Phase behavior in serial and parallel applications*, in 2012 IEEE International Symposium on Workload Characterization (IISWC), IEEE, 2012, pp. 47–58.

[5] A. Sembrant, D. Eklov, and E. Hagersten, *Efficient software-based online phase classification*, in Workload Characterization (IISWC), 2011 IEEE International Symposium on, IEEE, 2011, pp. 104–115.

[6] ——, *Scarphase*, 2012.

[7] Amazon, *AWS and Sustainability*, 2019.

[8] ——, *Amazon EMR Pricing*, 2020.

[9] Apache, *Apache Spark™ - Unified Analytics Engine for Big Data*, 2020.

[10] B. Gregg, *Systems performance: enterprise and the cloud*, Pearson Education, 2013.

[11] 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, 79 (2018), pp. 1–15.

[12] P. Belknap, B. Dageville, K. Dias, and K. Yagoub, *Self-tuning for sql performance in oracle database 11g*, in 2009 IEEE 25th International Conference on Data Engineering, IEEE, 2009, pp. 1694–1700.

[13] Carbon Fund Foundation, *Calculation Methods*, 2019.

[14] N. Chaimov, A. Malony, S. Canon, C. Iancu, K. Z. Ibrahim, and J. Srinivasan, *Scaling spark on hpc systems*, in Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing, 2016, pp. 97–110.

[15] 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.

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

[17] A. S. Dhodapkar and J. E. Smith, *Comparing program phase detection techniques*, in Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture, IEEE Computer Society, 2003, p. 217.

[18] S. Duan, V. Thummala, and S. Babu, *Tuning database configuration parameters with ituned*, Proceedings of the VLDB Endowment, 2 (2009), pp. 1246–1257.

[19] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder, *Using simpoint for accurate and efficient simulation*, in Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '03, New York, NY, USA, 2003, ACM, pp. 318–319.

[20] F. Bellard, *Qemu, a fast and portable dynamic translator*, in Proceedings of the Annual Conference on USENIX Annual Technical Conference, ATEC '05, Berkeley, CA, USA, 2005, USENIX Association, pp. 41–41.

[21] G. D. Israel, *Determining sample size*, (1992).

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

[23] 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.

[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] 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, 43 (2016), pp. 158–169.

[26] D. C. Knowledge, *The Facebook Data Center FAQ*, 2010.

[27] J. Laskowski, *The Internals of Apache Spark 2.4.2*.

[28] 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.

[29] A. Or, *Dynamic Allocation in Spark*, 2015.

[30] P. Dollár, Piotr and L. C. Zitnick, *Structured forests for fast edge detection*, in Proceedings of the IEEE International Conference on Computer Vision, 2013, pp. 1841–1848.

[31] P. Nagpurkar, P. Hind, C. Krintz, P. Sweeney, and V. Rajan, *Online phase detection algorithms*, in Code Generation and Optimization, 2006. CGO 2006. International Symposium on, IEEE, 2006, pp. 13–pp.

[32] 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, 12 (2011), pp. 2825–2830.

[33] 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.

[34] 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.

[35] Standard Performance Evaluation Corporation, *SPEC CPU® 2006*.

[36] ——, *SPEC CPU® 2017*.

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

[38] G. H. T. Sherwood, E. Perelman and B. . Calder, *Automatically characterizing large scale program behavior*, ACM SIGARCH Computer Architecture News, 30 (2002), pp. 45–57.

[39] T. Sherwood, E. Perelman, and B. Calder, *Basic block distribution analysis to find periodic behavior and simulation points in applications*, in Parallel Architectures and Compilation Techniques, 2001. Proceedings. 2001 International Conference on, IEEE, 2001, pp. 3–14.

[40] T. Sherwood, S. Sair, and B. Calder, *Phase tracking and prediction*, in ACM SIGARCH Computer Architecture News, vol. 31, ACM, 2003, pp. 336–349.

[41] 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.

[42] 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), 4 (2008), p. 3.

[43] S. Usman, R. Mehmood, and I. Katib, *Big data and hpc convergence: The cutting edge and outlook*, in International Conference on Smart Cities, Infrastructure, Technologies and Applications, Springer, 2017, pp. 11–26.

[44] 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.

[45] X. Shen, Y. Zhong, and C. Ding, *Locality phase prediction*, ACM SIGPLAN Notices, 39 (2004), pp. 165–176.

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

[47] 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, 59 (2016), pp. 56–65.

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

[49] 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.