A Zero-Positive Learning Approach for Diagnosing Software Performance Regressions

Mejbah Alam
Intel Labs
mejbah.alam@intel.com

Justin Gottschlich
Intel Labs
justin.gottschlich@intel.com

Nesime Tatbul
Intel Labs and MIT
tatbul@csail.mit.edu

Javier Turek
Intel Labs
javier.turek@intel.com

Timothy Mattson
Intel Labs
timothy.g.mattson@intel.com

Abdullah Muzahid
Texas A&M University
abdullah.muzahid@tamu.edu

Abstract

The field of *machine programming* (MP), the automation of the development of software, is making notable research advances. This is, in part, due to the emergence of a wide range of novel techniques in machine learning. In this paper, we apply MP to the automation of software performance regression testing. A *performance regression* is a software performance degradation caused by a code change. We present *AutoPerf* – a novel approach to automate regression testing that utilizes three core techniques: *(i)* zero-positive learning, *(ii)* autoencoders, and *(iii)* hardware telemetry. We demonstrate AutoPerf's generality and efficacy against 3 types of performance regressions across 10 real performance bugs in 7 benchmark and open-source programs. On average, AutoPerf exhibits 4% profiling overhead and accurately diagnoses more performance bugs than prior state-of-the-art approaches. Thus far, AutoPerf has produced no false negatives.

1 Introduction

Machine programming (MP) is the automation of the development and maintenance of software. Research in MP is making considerable advances, in part, due to the emergence of a wide range of novel techniques in machine learning and formal program synthesis [11, 12, 16, 37, 43, 44, 46, 47, 52, 55, 62]. A recent review paper proposed *Three Pillars of Machine Programming* as a framework for organizing research on MP [25]. These pillars are *intention*, *invention*, and *adaptation*.

Intention is concerned with simplifying and broadening the way a user's ideas are expressed to machines. *Invention* is the exploration of ways to automatically discover the right algorithms to fulfill those ideas. *Adaptation* is the refinement of those algorithms to function correctly, efficiently, and securely for a specific software and hardware ecosystem. In this paper, we apply MP to the automation of software testing, with a specific emphasis on parallel program performance regressions. Using the three pillars nomenclature, this work falls principally in the adaptation pillar.

Software performance regressions are defects that are erroneously introduced into software as it evolves from one version to the next. While they do not impact the functional correctness of the software, they can cause significant degradation in execution speed and resource efficiency (e.g., cache contention). From database systems to search engines to compilers, performance regressions are

33rd Conference on Neural Information Processing Systems (NeurIPS 2019), Vancouver, Canada.

commonly experienced by almost all large-scale software systems during their continuous evolution and deployment life cycle [7, 24, 30, 32, 34]. It may be impossible to entirely avoid performance regressions during software development, but with proper testing and diagnostic tools, the likelihood for such defects to silently leak into production code might be minimized.

Today, many benchmarks and testing tools are available to detect the presence of performance regressions [1, 6, 8, 17, 42, 57], but diagnosing their root causes still remains a challenge. Existing solutions either focus on whole program analysis rather than code changes [15], or depend on previously seen instances of performance regressions (i.e., rule-based or supervised learning approaches [20, 29, 33, 59]). Furthermore, analyzing multi-threaded programs running over highly parallel hardware is much harder due to the *probe effect* often incurred by traditional software profilers and debuggers [23, 26, 27]. Therefore, a more general, lightweight, and reliable approach is needed.

In this work, we propose AutoPerf, a new framework for software performance regression diagnostics, which fuses multiple state-of-the-art techniques from hardware telemetry and machine learning to create a unique solution to the problem. First, we leverage *hardware performance counters (HPCs)* to collect fine-grained information about run-time executions of parallel programs in a lightweight manner [10]. We then utilize *zero-positive learning (ZPL)* [36], *autoencoder neural networks* [60], and *k-means clustering* [35] to build a general and practical tool based on this data. Our tool, AutoPerf, can learn to diagnose potentially any type of regression, with minimal supervision.

We treat performance defects as anomalies that represent deviations from the normal behavior of a software program. Given two consecutive versions of a program P, P_i and P_{i+1}, the main task is to identify anomalies in P_{i+1}'s behavior with respect to the normal behavior represented by that of P_i. To achieve this, first we collect HPC profiles for functions that differ in P_i and P_{i+1}, by running each program with a set of test inputs. We then train autoencoder models using the profiles collected for P_i, which we test against the HPC profiles collected for P_{i+1}. Run instances where the autoencoder reconstruction error (RE) is above a certain threshold are classified as regressions. Finally, these regressions are analyzed to determine their types, causes, and locations in P_{i+1}.

Our framework enhances the state of the art along three dimensions:

- *Generality:* ZPL and autoencoders eliminate the need for labeled training data, while HPCs provide data on any detectable event. This enables our solution to generalize to any regression pattern.
- *Scalability:* Low-overhead HPCs are collected only for changed code, while training granularity can be adjusted via k-means clustering. This enables our solution to scale with data growth.
- *Accuracy:* We apply a statistical heuristic for thresholding the autoencoder reconstruction error, which enables our solution to identify performance defects with significantly higher accuracy.

In the rest of this paper, after some background, we first present our approach and then show the effectiveness of our solution with an experimental study on real-world benchmarks (PARSEC [17] and Phoenix [57] benchmark suites) and open-source software packages (Boost, Memcached, and MySQL). With only 4% average profiling overhead, our tool can successfully detect three types of performance regressions common in parallel software (true sharing, false sharing, and NUMA latency), at consistently higher accuracy than two state-of-the-art approaches [21, 33].

2 Motivation

Industrial software development is constantly seeking to accelerate the rate in which software is delivered. Due to the ever increasing frequency of deployments, software performance defects are leaking into production software at an alarming rate [34]. Because this trend is showing no sign of slowing, there is an increasing need for the practical adoption of techniques that automatically discover performance anomalies to prevent their integration to production-quality software [54]. To achieve this goal, we must first understand the challenges that inhibit building practical solutions. This section discusses such challenges and their potential solutions.

2.1 Challenges: Diagnosing Software Performance Regressions

Detailed software performance diagnostics are hard to capture. We see two core challenges.

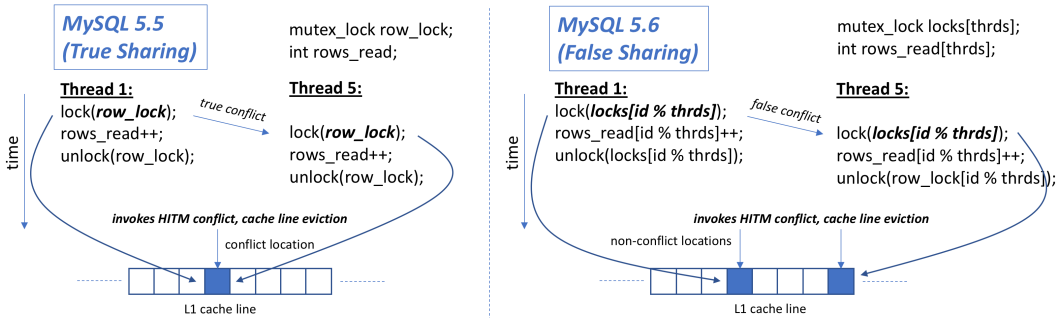


Figure 1: Example of performance regressions in parallel software.

Examples are limited. Software performance regressions can manifest in a variety of forms and frequencies. Due to this, it is practically impossible to exhaustively identify all of them a priori. In contrast, normal performance behaviors are significantly easier to observe and faithfully capture.

Profiling may perturb performance behavior. Software profiling via code instrumentation may cause perturbations in a program's run-time behavior. This is especially true for parallel software, where contention signatures can be significantly altered due to the most minute *probe effect* [26, 48] (e.g., a resource contention defect may become unobservable).

These challenges call for an approach that *(i)* does not rely on training data that includes performance regressions and *(ii)* uses a profiling technique which incurs minimal execution overhead (i.e., less than 5%) as to not perturb a program's performance signature. Next, we provide concrete examples of performance bugs that are sensitive to these two criteria.

2.2 Examples: Software Performance Regressions

Cache contention may occur when multiple threads of a program attempt to access a shared memory cache concurrently. It comes in two flavors: (i) *true sharing*, involving access to the same memory location, and (ii) *false sharing*, involving access to disjoint memory locations on the same cache line. For example, a true sharing defect in MySQL 5.5 is shown in Figure 1(a). Unfortunately, developer's attempt to fix this issue could cause a performance regression due to false sharing defect. This defect in Figure 1(b) was introduced into MySQL version 5.6, leading to more than a 67% performance degradation [9].

NUMA latency may arise in Non-Uniform Memory Access (NUMA) architectures due to a mismatch between where data is placed in memory vs. the CPU threads accessing it. For example, the streamcluster application of the PARSEC benchmark was shown to experience a 25.7% overall performance degradation due to NUMA [17].

These types of performance defects are generally challenging to identify from source code. An automatic approach can leverage HPCs as a feature to identify these defects (more in Section 4.2).

2.3 A New Approach: Zero-Positive Learning Meets Hardware Telemetry

To address the problem, we propose a novel approach that consists of two key ingredients: zero-positive learning (ZPL) [36] and hardware telemetry [10].

ZPL is an implicitly supervised ML technique. It is a specific instance of one-class classification, where all training data lies within one class (i.e., the non-anomalous space). ZPL was originally developed for anomaly detection (AD). In AD terminology, a positive refers to an anomalous data sample, while a negative refers to a normal one, thus the name *zero-positive learning*. Any test data that sufficiently deviates from the negative distribution is deemed an anomaly. Thus, ZPL, if coupled with the right ML modeling technique, can provide a practical solution to the first challenge, as it does not require anomalous data.

Hardware telemetry enables profiling program executions using hardware performance counters (HPCs). HPCs are a set of special-purpose registers built into CPUs to store counts of a wide range of hardware-related activities, such as instructions executed, cycles elapsed, cache hits or misses, branch (mis)predictions, etc. Modern-day processors provide hundreds of HPCs, and more are being added

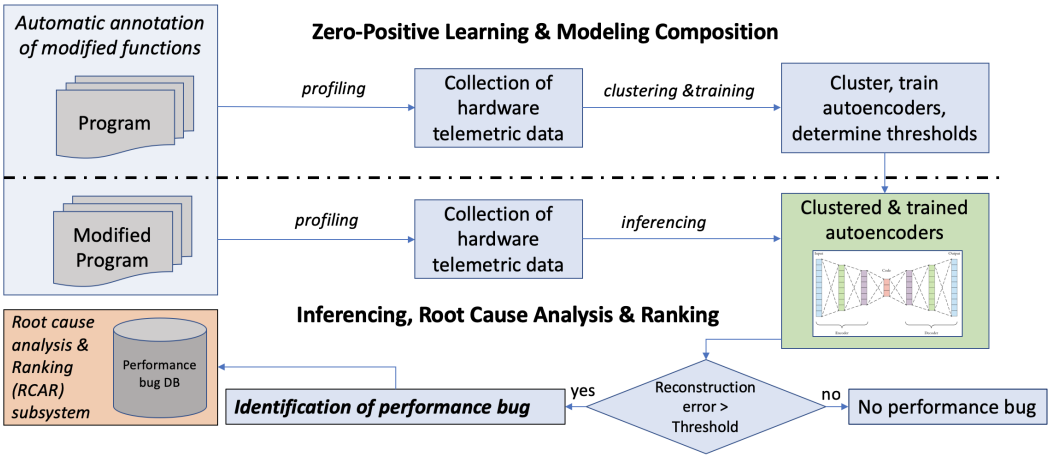


Figure 2: Overview of AutoPerf

with every new architecture. As such, HPCs provide a lightweight means for collecting fine-grained profiling information without modifying source code, addressing the second challenge.

3 Related Work

There has been an extensive body of prior research in software performance analysis using statistical and ML techniques [15, 31, 38, 53, 58]. Most of the past ML approaches are based on traditional supervised learning models (e.g., Bayesian networks [20, 59], Markov models [29], decision trees [33]). A rare exception is the unsupervised behavior learning (UBL) approach of Dean et al., which is based on self-organizing maps [21]. Unfortunately, UBL does not perform well beyond a limited number of input features. To the best of our knowledge, ours is the first scalable ML approach for software performance regression analysis that relies only on normal (i.e., no example of performance regression) training data.

Prior efforts commonly focus on analyzing a specific type of performance defect (e.g., false and/or true sharing cache contention [22, 33, 39–41, 51, 63], NUMA defects [42, 56, 61]). Some of these also leverage HPCs like we do [14, 18, 28, 40, 56, 61]. However, our approach is general enough to analyze any type of performance regression based on HPCs, including cache contention and NUMA latency. Overall, the key difference of our contribution lies in its practicality and generality. Section 5 presents an experimental comparison of our approach against two of the above approaches [21, 33].

4 The Zero-Positive Learning Approach

In this section, we present a high-level overview of our approach, followed by a detailed discussion of its important and novel components.

4.1 Design Overview

A high-level design of AutoPerf is shown in Figure 2. Given two versions of a software program, AutoPerf first compares their performance. If a degradation is observed, then the cause is likely to lie within the functions that differ in the two versions. Hence, AutoPerf automatically annotates the modified functions in both versions of the program and collects their HPC profiles. The data collected for the older version is used for zero-positive model training, whereas the data collected for the newer version is used for inferencing based on the trained model. AutoPerf uses an autoencoder neural network to model normal performance behavior of a function [60]. To scale with a large number of functions, training data for functions with similar performance signatures are clustered together using k-means clustering and a single autoencoder model per cluster is trained [35]. Performance regressions are identified by measuring the reconstruction error that results from testing the autoencoders with profile data from the new version of the program. If the error comes out to be sufficiently high, then

the corresponding execution of the function is marked as a performance bug and its root cause is analyzed as the final step of the diagnosis.

4.2 Data Collection

Modern processors provide various hardware performance counters (HPCs) to count low-level system events such as cache misses, instruction counts, memory accesses [10]. AutoPerf uses Performance Application Programming Interface (PAPI) to read values of hardware performance counters [49]. For example, for the specific hardware platform that we used in our experimental work (see Section 5.1 for details), PAPI provides access to 50 different HPCs. Many of these performance counters reflect specific performance features of a program running on the specific hardware. For example, Hit Modified (HITM) is closely related to cache contention [45]. Essentially, this counter is incremented every time a processor accesses a memory cache line which is modified in another processor's cache. Any program with true or false sharing defects will see a significant increase in the HITM counter's value. Similarly, the counter for off-core requests served by remote DRAM (OFFCORE_RESPONSE: REMOTE_DRAM) can be used to identify NUMA-related performance defects [42]. AutoPerf exploits these known features in its final root-cause analysis step.

To collect HPC profiles of all modified functions, we execute both of the annotated program versions with a set of test inputs (i.e., regression test cases). Test inputs generally capture a variety of different input sizes and thread counts. During each execution of an annotated function `foo`, AutoPerf reads HPCs at both the entry and the exit points of `foo`, calculates their differential values, normalizes these values with respect to the instruction count of `foo` and thread count, and records the resulting values as one sample in `foo`'s HPC profile.

4.3 Diagnosing Performance Regressions

AutoPerf uses HPC profiles to diagnose performance regressions in a modified program. First, it learns the distribution of the performance of a function based on its HPC profile data collected from the original program. Then, it detects deviations of performance as anomalies based on the HPC profile data collected from the modified program.

4.3.1 Autoencoder-based Training and Inference

Our approach to performance regression automation requires to solve a zero-positive learning task. Zero-positive learning involves a one-class training problem, where only negative (non-anomalous) samples are used at training time [50]. We employ autoencoders to learn the data distribution of the non-anomalous data [13]. At test time, we then exploit the autoencoder to discover any deviation that would indicate a sample from the positive class. The autoencoder model is a natural fit for our ZPL approach, since it is unsupervised (i.e., does not require labeled training data as in one-class training) and it works well with multi-dimensional inputs (i.e., data from multiple HPCs).

To formalize, let $\{\mathbf{x}_i\}_{i=1}^{N_{old}}$ be a set of N_{old} samples obtained from profiling the old version of the function `foo`. Next, we train an autoencoder $\mathcal{A}_{\text{foo}}(\mathbf{x}) = f(g(\mathbf{x}))$ such that it minimizes the reconstruction error over all samples, i.e., $\mathcal{L}(\mathbf{x}_i, \mathcal{A}_{\text{foo}}(\mathbf{x}_i)) = \sum_i \|\mathbf{x}_i - \mathcal{A}_{\text{foo}}(\mathbf{x}_i)\|_2^2$. During training, the autoencoder $\mathcal{A}_{\text{foo}}(\mathbf{x})$ learns a manifold embedding represented by its encoder $g(\mathbf{x})$. Its decoder $f(\mathbf{x})$ learns the projection back to sample space. Learning the manifold embedding is crucial to the autoencoder to reconstruct a sample with high fidelity.

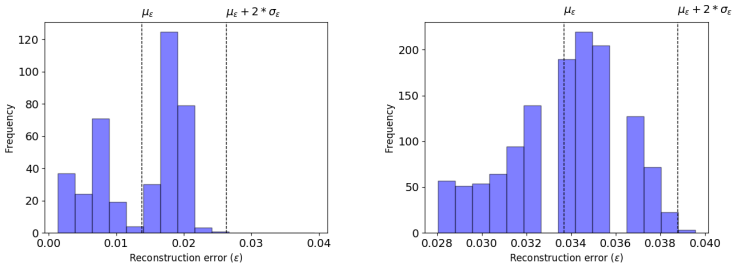
Once the autoencoder is trained, AutoPerf collects an additional set of samples $\{\mathbf{z}_i\}_{i=1}^{N_{new}}$ profiling the newer version of function `foo`'s code. Next, the system discovers anomalies by encoding and decoding the new samples $\mathbf{z}_i$ and measuring the reconstruction error, i.e.,

$$\epsilon(\mathbf{z}_i) = \|\mathbf{z}_i - \mathcal{A}_{\text{foo}}(\mathbf{z}_i)\|_2 \tag{1}$$

If the reconstruction error for a sample $\mathbf{z}_i$ is above a certain threshold γ, i.e., $\epsilon > \gamma$, the sample is marked as anomalous, as it lays sufficiently distant from its back-projected reconstruction.

4.3.2 Reconstruction Error Threshold Heuristic

The success to detect the anomalous samples heavily depends on setting the right value for threshold γ. A value too high and we may fail to detect many anomalous samples, raising the number of false



(a) dedup [17] (b) MySQL [9]

Figure 3: Histograms of reconstruction error ϵ for training samples $\{\mathbf{x}_i\}$ of two real datasets

negatives. A value too low, and AutoPerf will detect many non-anomalous samples as anomalous, increasing the number of false positives. Figure 3(a) and (b) show the reconstruction errors for the training samples for the dedup and MySQL datasets, respectively [9, 17]. Clearly, the difference in histograms signals that naïvely setting a threshold would not generalize across datasets or even functions.

The skewness in the reconstruction error distributions of all test applications ranges from -0.46 to 0.08. The kurtosis ranges from 1.96 to 2.64. Therefore, we approximate the reconstruction error's distribution with a Normal distribution and define a threshold $\gamma(t)$ relative to the errors as

$$\gamma(t) = \mu_\epsilon + t\sigma_\epsilon \tag{2}$$

where μ_ϵ is the mean reconstruction error and σ_ϵ its standard deviation for the training samples $\{\mathbf{x}_i\}$. The t parameter controls the level of thresholding. For example, with $t = 2$, the threshold provides (approximately) a 95% confidence interval for the reconstruction error.

To find the cause (and type) of performance regression, we calculate the reconstruction error (RE) for each performance counter corresponding to an anomalous sample and then, sort the counters accordingly. We take a majority vote among all performance counters for each anomalous sample. The counter that comes first is the one that causes the performance regression. We report that counter and the corresponding regression type as the root cause.

4.3.3 Scaling to Many Functions via k-means Clustering

So far, we have focused on analyzing the performance of a single function that is modified with new code. In reality, the number of functions that change between versions of the code is higher. For example, 27 functions are modified between two versions of MySQL used in our experiments [9]. Training one autoencoder per such function is impractical. Furthermore, the number of samples required to train these grows, too. To alleviate this, we group multiple functions into clusters and assign an autoencoder to each group. AutoPerf applies k-means clustering for this purpose [35]. It computes k clusters from the training samples. Then, we assign function f to cluster c, if c contains more samples of f than any other cluster. For each cluster c, we build one autoencoder. We train the autoencoder using the training samples of all the functions that belong to that cluster. During inferencing, when we analyze profiling samples for a newer version of a function, we feed them to the autoencoder of the cluster where that function belongs to.

5 Experimental Evaluation

In this section, we *(i)* evaluate AutoPerf's ability to diagnose performance regressions and compare with two state-of-the-art machine learning based approaches: Jayasena et al. [33] and UBL [21], *(ii)* analyze our clustering approach, and *(iii)* quantify profiling and training overheads.

5.1 Experimental Setup

We used PAPI to read hardware performance counter values [49], and Keras with TensorFlow to implement autoencoders [19]. PAPI provides a total of 50 individualized and composite HPCs. We read the 33 individualized counters during profiling as input features to AutoPerf. We performed all

Table 1: Diagnosis ability of AutoPerf vs. DT [33] and UBL [21]. *TS* = True Sharing, *FS* = False Sharing, and *NL* = NUMA Latency. K, L, M are the # of executions ($K = 6, L = 10, M = 20$).

Normal Program	False Positive Rate			Anomalous Program	Defect Type	False Negative Rate		
	AutoPerf	DT	UBL			AutoPerf	DT	UBL
blackscholes$_L$	0.0	N/A	0.2	blackscholes$_K$	*NL*	0.0	N/A	0.0
bodytrack$_L$	0.0	0.7	0.8	bodytrack$_K$	*TS*	0.0	0.17	0.1
dedup$_L$	0.0	1.0	0.2	dedup$_K$	*TS*	0.0	0.0	0.0
histogram$_M$	0.0	0.0	0.0	histogram$_M$	*FS*	0.0	0.1	1.0
linear_regression$_M$	0.0	0.3	0.0	linear_regression$_M$	*FS*	0.0	0.4	0.35
reverse_index$_M$	0.0	0.4	0.15	reverse_index$_M$	*FS*	0.0	0.1	0.05
streamcluster$_L$	0.0	N/A	0.6	streamcluster$_K$	*NL*	0.0	N/A	0.1
Boost$_L$	**0.3**	1.0	0.4	Boost$_L$	*FS*	0.0	0.2	0.2
Memcached$_L$	0.0	1.0	0.4	Memcached$_L$	*TS*	0.0	0.4	0.3
MySQL$_L$	**0.2**	1.0	0.1	MySQL$_L$	*FS*	0.0	0.5	0.8

experiments on a 12-core dual socket Intel Xeon© Scalable 8268 processor [3] with 32GB RAM. We used 7 programs with known performance defects from the PARSEC [17] and the Phoenix [57] benchmark suites. Additionally, we evaluated 3 open-source programs: Boost [2], Memcached [4], and MySQL [5].

5.2 Diagnosis Ability

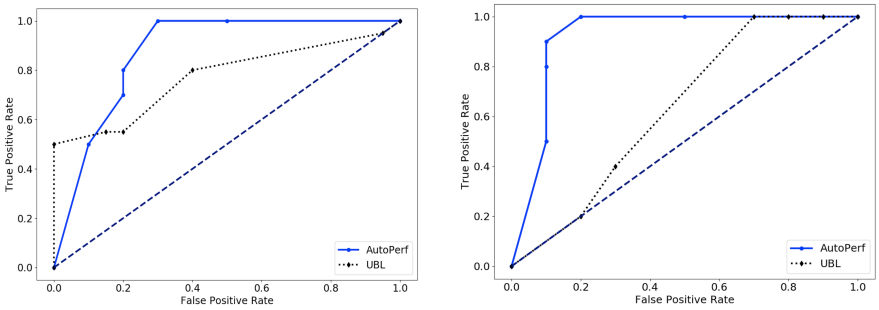
We experiment with 10 programs to evaluate AutoPerf. Two versions of source code for each program are used for these experiments: 1) a version without any performance defect; 2) a version where a performance defect is introduced after updating one or more functions in the first version. We run the first version n number of times. If a system reports x number of these runs as anomalous (i.e., positive), we define *false positive rate* as x/n. Similarly, we run the second version m number of times and define *false negative rate* as x/m, where x is the number of anomalous runs detected as non-anomalous. Each run of a program uses different inputs.

AutoPerf's diagnosis results are summarized in Table 1. We experimented with 3 different types of performance defects across 7 benchmark programs and 3 real-world applications. These are known performance bugs (confirmed by developers) in real-world and benchmark applications. A version of each application, for which corresponding performance defect is reported, is used for generating anomalous runs. AutoPerf detects performance defects in all anomalous runs. However, it reports 3 false positive runs in Boost and 2 false positive runs in MySQL. Anomalies in Boost are detected in a function that implements a *spinlock*. It implements lock acquisition by iteratively trying to acquire the lock within a loop. Moreover, these runs are configured with increased number of threads. We suspect that these false positive test runs experienced increased lock contention, which was not present in training runs. This could be improved by increasing the variability of inputs for training runs. The two false positive runs in MySQL are reported in two functions. These are small functions with reduced number of instructions, which could affect the accuracy of profiling at a fixed sampling rate.

We quantitatively compared AutoPerf with two state-of-the-art machine learning based approaches: Jayasena et al. [33] and UBL [21]. Jayasena et al. uses a decision tree of 12 performance counters to detect true sharing and false sharing defects (DT in Table 1). This approach is limited to detection of false sharing and true sharing types of performance detect. Therefore, it cannot detect the NUMA performance defects in blackscholes and streamcluster. Moreover, [33] uses a fixed ratio of various counters and therefore, cannot detect all anomalous runs in 6 programs and reports false positive runs for all 8 programs.

We implemented UBL using a 120×120 self-organizing map (SOM) to detect performance anomalies. Table 1 shows UBL reports greater number of false positive runs for 7 programs and greater false negative runs for 7 programs. The reduction in accuracy is caused by SOM's limitation in handling large variations in performance counter values. Overall, AutoPerf produces false positives for Boost and MySQL, whereas other approaches produces false positives or false negatives nearly for every program. We further evaluated the anomaly prediction accuracy of AutoPerf using the standard

receiver operating characteristic (ROC) curves. Figure 4 shows ROC curves for Boost and MySQL. Although AutoPerf produces false positives for these two applications, the ROC curves show that it achieves better accuracy than UBL for these two applications.



(a) (b)

Figure 4: Diagnosis of false sharing defects in (a) Boost and (b) MySQL. True positive rates and false positive rates of AutoPerf and state-of-the-art approach UBL [21] for an application with different thresholds are shown in each figure.

5.3 Impact of Clustering

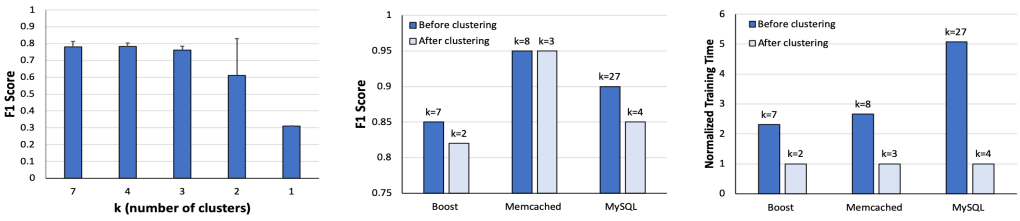
To analyze many functions that change between versions of code with reduced number of autoencoders, AutoPerf combines groups of similar functions into clusters and train an autoencoder for each cluster. We experimented with AutoPerf's accuracy to evaluate if the clustering reduces the accuracy of the system compared to using one autoencoder for each function.

One way to evaluate this is to test it against a program with multiple performance defects in different functions. To achieve this, we performed a sensitivity analysis using a synthetic program constructed with seven functions. We created a modified version of this program by introducing performance defects in each of these functions and evaluated the F_1 score of AutoPerf with different number of clusters for these seven functions in the program. AutoPerf achieves a reasonable F_1 score (from 0.73 to 0.81) using one autoencoder per function. When it uses one autoencoder across all seven functions, F_1 degrades significantly to 0.31. Using k-means clustering we can achieve reasonable accuracy even without one autoencoder per function. As shown in Figure 5(a), there is an increase in accuracy (F_1 score) as k increases from 2 to 3 to 4.

We evaluate the effects of clustering in three real-world programs: Boost, Memcached, and MySQL. Figure 5(b) shows accuracy of these programs using F_1 score. For Memcached, AutoPerf creates three clusters from eight candidate functions (i.e., changed functions). The F_1 score after clustering becomes equal to the F_1 score of an approach that uses one autoencoder per function. For other two programs: Boost and MySQL, clustering results in slightly reduced F_1 score. However, as shown in Figure 5(c), the clustering approach reduces overall training time of AutoPerf by 2.5x to 5x.

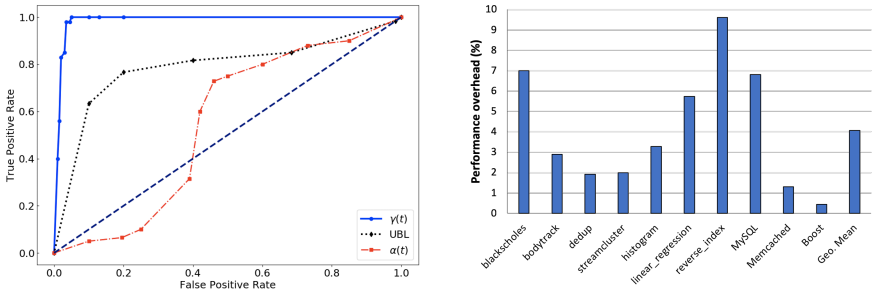
5.4 Effectiveness of the Error Threshold

We evaluated the effectiveness of our threshold method for $\gamma(t)$. We compared with a base approach of setting an arbitrary threshold based on the input vector x instead of reconstruction errors. This arbitrary threshold, $\alpha(t)$, implies that if the difference between the output and input vector length is more than $t\%$ of the input vector length x, it is marked as anomalous. We compared accuracy of AutoPerf with UBL and this base approach using the mean true positive rates and mean false positive rates of these approaches across 10 candidate applications listed in Table 1. Figure 6(a) shows the accuracy of AutoPerf using arbitrary threshold and $\gamma(t)$. We evaluated AutoPerf with different thresholds determined using equation (2), where values of t ranges from 0 to 3. AutoPerf achieves true positive rate of 1 and false positive rate of 0.05 using $\gamma(t)$ at $t = 2$. For arbitrary threshold using $\alpha(t)$, we experimented with increasing values of t from 0 to 55, at which point both true positive rate and false positive rate become 1. Figure 6 also shows the accuracy of UBL with different thresholds. $\gamma(t)$ achieves increased accuracy compared to UBL and $\alpha(t)$. Moreover, $\alpha(t)$ performs even worse than the best results from UBL.



(a) Sensitivity to k (b) Impact on accuracy (c) Impact on training time

Figure 5: Impact of clustering, where k denotes the number of clusters (i.e., autoencoders)



(a) (b)

Figure 6: (a) Effect of error threshold, (b) Profiling overhead.

5.5 Profiling and Training Overheads

Profiling of a program introduces performance overhead. However, AutoPerf uses HPCs to implement a lightweight profiler. The execution time of an application increases by only 4%, on average, with AutoPerf. MySQL experiments results in the highest performance overhead of 7% among three real-world applications. AutoPerf monitors greater number of modified functions in MySQL compared to the other two real-world applications: Memcached and Boost. We also collected the training time of autoencoders. On average, it takes approximately 84 minutes to train an autoencoder. An autoencoder for MySQL, which models a cluster with many functions, takes the longest training time, which is little less than 5 hours using our experimental setup (Section 5.1).

6 Conclusion

In this paper, we presented AutoPerf, a generalized software performance analysis system. For learning, it uses a fusion of zero-positive learning, k-means clustering, and autoencoders. For features, it uses hardware telemetry in the form of hardware performance counters (HPCs). We showed that this design can effectively diagnose some of the most complex software performance bugs, like those hidden in parallel programs. Although HPCs are useful to detect performance defects with minimal perturbation, it can be challenging to identify the root cause of such bugs with HPCs alone. Further investigation into a more expressive program abstraction, coupled with our zero-positive learning approach, could pave the way for better root cause analysis. With better root cause analysis, we might be able to realize an automatic defect correction system for such bugs.

Acknowledgments

We thank Jeff Hammond for his suggestions regarding experimental setup details. We thank Mostofa Patwary for research ideas in the early stages of this work. We thank Pradeep Dubey for general research guidance and continuous feedback. We also thank all the anonymous reviewers and area chairs for their excellent feedback and suggestions that have helped us improve this work.

References

[1] Apache HTTP server benchmarking tool. https://httpd.apache.org/docs/2.4/programs/ab.html.

[2] Boost C++ Library. https://www.boost.org/.

[3] Intel Xeon Platinum 8268 Processor. https://ark.intel.com/.

[4] Memcached: A Distributed Memory Object Caching System. https://memcached.org/.

[5] MySQL Database. http://www.mysql.com/.

[6] SysBench Benchmark Tool. https://dev.mysql.com/downloads/benchmarks.html.

[7] MySQL bug 16504. https://bugs.mysql.com/bug.php?id=16504, 2006.

[8] Visual Performance Analyzer. ftp://ftp.software.ibm.com/aix/tools/perftools/SystempTechUniv2006/UnixLasVegas2006-A09.pdf, 2006.

[9] Bug 79454::Inefficient InnoDB row stats implementation. https://bugs.mysql.com/bug.php?id=79454, 2015.

[10] IA-32 Architectures Software Developers Manual Volume 3b System Programming Guide, part 2. Intel Manual, September 2016.

[11] A. Adams, K. Ma, L. Anderson, R. Baghdadi, T.-M. Li, M. Gharbi, B. Steiner, S. Johnson, K. Fatahalian, F. Durand, and J. Ragan-Kelley. Learning to Optimize Halide with Tree Search and Random Programs. *ACM Trans. Graph.*, 38(4):121:1–121:12, July 2019.

[12] M. B. S. Ahmad, J. Ragan-Kelley, A. Cheung, and S. Kamil. Automatically Translating Image Processing Libraries to Halide. *ACM Transactions on Graphics*, 38(6), Nov 2019.

[13] G. Alain and Y. Bengio. What Regularized Auto-Encoders Learn from the Data-Generating Distribution. *Journal of Machine Learning Research*, 15:3743–3773, 2014.

[14] J. Arulraj, P.-C. Chang, G. Jin, and S. Lu. Production-run Software Failure Diagnosis via Hardware Performance Counters. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 101–112, New York, NY, USA, 2013. ACM.

[15] M. Attariyan, M. Chow, and J. Flinn. X-ray: Automating Root-cause Diagnosis of Performance Anomalies in Production Software. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 307–320, Berkeley, CA, USA, 2012. USENIX Association.

[16] K. Becker and J. Gottschlich. AI Programmer: Autonomously Creating Software Programs Using Genetic Algorithms. *CoRR*, abs/1709.05703, 2017.

[17] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 72–81, New York, NY, USA, 2008. ACM.

[18] M. Brocanelli and X. Wang. Hang Doctor: Runtime Detection and Diagnosis of Soft Hangs for Smartphone Apps. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 6:1–6:15, New York, NY, USA, 2018. ACM.

[19] F. Chollet et al. Keras. https://github.com/fchollet/keras, 2015.

[20] I. Cohen, M. Goldszmidt, T. Kelly, J. Symons, and J. S. Chase. Correlating Instrumentation Data to System States: A Building Block for Automated Diagnosis and Control. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 16–16, Berkeley, CA, USA, 2004. USENIX Association.

[21] D. J. Dean, H. Nguyen, and X. Gu. UBL: Unsupervised Behavior Learning for Predicting Performance Anomalies in Virtualized Cloud Systems. In *Proceedings of the 9th International Conference on Autonomic Computing*, ICAC '12, pages 191–200, New York, NY, USA, 2012. ACM.

[22] A. Eizenberg, S. Hu, G. Pokam, and J. Devietti. Remix: Online Detection and Repair of Cache Contention for the JVM. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 251–265, New York, NY, USA, 2016. ACM.

[23] J. Gait. A Probe Effect in Concurrent Programs. *Software Practice and Experience*, 16(3):225–233, March 1986.

[24] T. Glek. Massive Performance Regression From Switching to GCC 4.5. `http://gcc.gnu.org/ml/gcc/2010-06/msg00715.html`.

[25] J. Gottschlich, A. Solar-Lezama, N. Tatbul, M. Carbin, M. Rinard, R. Barzilay, S. Amarasinghe, J. B. Tenenbaum, and T. Mattson. The Three Pillars of Machine Programming. In *Proceedings of the 2Nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2018, pages 69–80, New York, NY, USA, 2018. ACM.

[26] J. E. Gottschlich, M. P. Herlihy, G. A. Pokam, and J. G. Siek. Visualizing Transactional Memory. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 159–170, New York, NY, USA, 2012. ACM.

[27] J. E. Gottschlich, G. A. Pokam, C. L. Pereira, and Y. Wu. Concurrent Predicates: A Debugging Technique for Every Parallel Programmer. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, pages 331–340, Piscataway, NJ, USA, 2013. IEEE Press.

[28] J. L. Greathouse, Z. Ma, M. I. Frank, R. Peri, and T. Austin. Demand-driven Software Race Detection Using Hardware Performance Counters. In *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, pages 165–176, June 2011.

[29] X. Gu and H. Wang. Online Anomaly Prediction for Robust Cluster Systems. In *Proceedings of the 2009 IEEE International Conference on Data Engineering*, ICDE '09, pages 1000–1011, Washington, DC, USA, 2009. IEEE Computer Society.

[30] S. Han, Y. Dang, S. Ge, D. Zhang, and T. Xie. Performance Debugging In The Large via Mining Millions of Stack Traces. In *ICSE*, pages 145–155. IEEE, 2012.

[31] L. Huang, J. Jia, B. Yu, B.-G. Chun, P. Maniatis, and M. Naik. Predicting Execution Time of Computer Programs Using Sparse Polynomial Regression. In *Proceedings of the 23rd International Conference on Neural Information Processing Systems - Volume 1*, NIPS'10, pages 883–891, USA, 2010. Curran Associates Inc.

[32] P. Huang, X. Ma, D. Shen, and Y. Zhou. Performance Regression Testing Target Prioritization via Performance Risk Analysis. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 60–71. ACM, 2014.

[33] S. Jayasena, S. Amarasinghe, A. Abeyweera, G. Amarasinghe, H. D. Silva, S. Rathnayake, X. Meng, and Y. Liu. Detection of False Sharing Using Machine Learning. In *2013 SC - International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–9, Nov 2013.

[34] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu. Understanding and Detecting Real-world Performance Bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 77–88, New York, NY, USA, 2012. ACM.

[35] T. Kanungo, D. M. Mount, N. S. Netanyahu, C. D. Piatko, R. Silverman, and A. Y. Wu. An Efficient k-Means Clustering Algorithm: Analysis and Implementation. *IEEE Trans. Pattern Anal. Mach. Intell.*, 24(7):881–892, July 2002.

[36] T. J. Lee, J. Gottschlich, N. Tatbul, E. Metcalf, and S. Zdonik. Greenhouse: A Zero-Positive Machine Learning System for Time-Series Anomaly Detection. In *Inaugural Conference on Systems and Machine Learning (SysML'18)*, Stanford, CA, USA, February 2018.

[37] C. Lemieux, R. Padhye, K. Sen, and D. Song. PerfFuzz: Automatically Generating Pathological Inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, pages 254–265, New York, NY, USA, 2018. ACM.

[38] J. Li, Y. Chen, H. Liu, S. Lu, Y. Zhang, H. S. Gunawi, X. Gu, X. Lu, and D. Li. Pcatch: Automatically Detecting Performance Cascading Bugs in Cloud Systems. In *Proceedings of*

the Thirteenth EuroSys Conference, EuroSys '18, pages 7:1–7:14, New York, NY, USA, 2018. ACM.

[39] T. Liu and E. D. Berger. SHERIFF: Precise Detection and Automatic Mitigation of False Sharing. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 3–18, New York, NY, USA, 2011. ACM.

[40] T. Liu and X. Liu. Cheetah: Detecting False Sharing Efficiently and Effectively. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization (CGO'16)*, pages 1–11, 2016.

[41] T. Liu, C. Tian, Z. Hu, and E. D. Berger. PREDATOR: Predictive False Sharing Detection. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 3–14, New York, NY, USA, 2014. ACM.

[42] X. Liu and J. Mellor-Crummey. A Tool to Analyze the Performance of Multithreaded Programs on NUMA Architectures. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 259–272, New York, NY, USA, 2014. ACM.

[43] C. Loncaric, M. D. Ernst, and E. Torlak. Generalized Data Structure Synthesis. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 958–968, New York, NY, USA, 2018. ACM.

[44] S. Luan, D. Yang, C. Barnaby, K. Sen, and S. Chandra. Aroma: Code Recommendation via Structural Code Search. *Proc. ACM Program. Lang.*, 3(OOPSLA):152:1–152:28, Oct. 2019.

[45] L. Luo, A. Sriraman, B. Fugate, S. Hu, G. Pokam, C. J. Newburn, and J. Devietti. LASER: Light, Accurate Sharing dEtection and Repair. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 261–273, March 2016.

[46] S. Mandal, T. A. Anderson, J. Gottschlich, S. Zhou, and A. Muzahid. Learning Fitness Functions for Genetic Algorithms, 2019.

[47] R. Marcus, P. Negi, H. Mao, C. Zhang, M. Alizadeh, T. Kraska, O. Papaemmanouil, and N. Tatbul. Neo: A Learned Query Optimizer. *Proc. VLDB Endow.*, 12(11):1705–1718, July 2019.

[48] C. E. McDowell and D. P. Helmbold. Debugging Concurrent Programs. *ACM Comput. Surv.*, 21(4):593–622, Dec. 1989.

[49] S. V. Moore. A Comparison of Counting and Sampling Modes of Using Performance Monitoring Hardware. In *In International Conference on Computational Science (ICCS 2002*, 2002.

[50] M. M. Moya and D. R. Hush. Network Constraints and Multi-objective Optimization for One-class Classification. *Neural Networks*, 9(3):463 – 474, 1996.

[51] M. Nanavati, M. Spear, N. Taylor, S. Rajagopalan, D. T. Meyer, W. Aiello, and A. Warfield. Whose Cache Line is It Anyway?: Operating System Support for Live Detection and Repair of False Sharing. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 141–154, New York, NY, USA, 2013. ACM.

[52] L. Nelson, J. Bornholt, R. Gu, A. Baumann, E. Torlak, and X. Wang. Scaling symbolic evaluation for automated verification of systems code with Serval. In *27th ACM Symposium on Operating Systems Principles (SOSP)*. ACM, October 2019.

[53] T. H. Nguyen, B. Adams, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora. Automated Detection of Performance Regressions Using Statistical Process Control Techniques. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, ICPE '12, pages 299–310, New York, NY, USA, 2012. ACM.

[54] A. Nistor, T. Jiang, and L. Tan. Discovering, Reporting, and Fixing Performance Bugs. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 237–246, May 2013.

[55] P. M. Phothilimthana, A. S. Elliott, A. Wang, A. Jangda, B. Hagedorn, H. Barthels, S. J. Kaufman, V. Grover, E. Torlak, and R. Bodik. Swizzle Inventor: Data Movement Synthesis for GPU Kernels. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 65–78, New York, NY, USA, 2019. ACM.

[56] A. Rane and J. Browne. Enhancing Performance Optimization of Multicore Chips and Multichip Nodes with Data Structure Metrics. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 147–156, New York, NY, USA, 2012. ACM.

[57] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 13–24, Washington, DC, USA, 2007. IEEE Computer Society.

[58] L. Song and S. Lu. Statistical Debugging for Real-world Performance Problems. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 561–578, New York, NY, USA, 2014. ACM.

[59] Y. Tan, H. Nguyen, Z. Shen, X. Gu, C. Venkatramani, and D. Rajan. PREPARE: Predictive Performance Anomaly Prevention for Virtualized Cloud Systems. In *Proceedings of the 2012 IEEE 32Nd International Conference on Distributed Computing Systems*, ICDCS '12, pages 285–294, Washington, DC, USA, 2012. IEEE Computer Society.

[60] P. Vincent, H. Larochelle, Y. Bengio, and P.-A. Manzagol. Extracting and Composing Robust Features with Denoising Autoencoders. In *Proceedings of the 25th International Conference on Machine Learning*, ICML '08, pages 1096–1103, New York, NY, USA, 2008. ACM.

[61] R. Yang, J. Antony, A. Rendell, D. Robson, and P. Strazdins. Profiling Directed NUMA Optimization on Linux Systems: A Case Study of the Gaussian Computational Chemistry Code. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, IPDPS '11, pages 1046–1057, Washington, DC, USA, 2011. IEEE Computer Society.

[62] X. Zhang, A. Solar-Lezama, and R. Singh. Interpreting Neural Network Judgments via Minimal, Stable, and Symbolic Corrections. In *Proceedings of the 32Nd International Conference on Neural Information Processing Systems*, NIPS'18, pages 4879–4890, USA, 2018. Curran Associates Inc.

[63] Q. Zhao, D. Koh, S. Raza, D. Bruening, W.-F. Wong, and S. Amarasinghe. Dynamic Cache Contention Detection in Multi-threaded Applications. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '11, pages 27–38, New York, NY, USA, 2011. ACM.