Vision Paper

Stitcher: Learned Workload Synthesis from Historical Performance Footprints

Chengcheng Wan[1*], Yiwen Zhu[2], Joyce Cahoon[2], Wenjing Wang[2], Katherine Lin[2], Sean Liu[2], Raymond Truong[2], Neetu Singh[2], Alexandra Ciortea[2], Konstantinos Karanasos[2], Subru Krishnan[2]
[1] University of Chicago, [2] Microsoft
cwan@uchicago.edu,{<name>.<surname>}@microsoft.com

ABSTRACT

Database benchmarking and workload replay have been widely used to drive system design, evaluate workload performance, determine product evolution, and guide cloud migration. However, they both suffer from some key limitations: the former fails to capture the variety and complexity of production workloads; the latter requires access to user data, queries, and machine specifications, deeming it inapplicable in the face of user privacy concerns. Here we introduce our vision of *learned workload synthesis* to overcome these issues: given the performance profile of a customer workload (e.g., CPU/memory counters), synthesize a new workload that yields the same performance profile when executed on a range of hardware/software configurations. We present *Stitcher* as a first step towards realizing this vision, which synthesizes workloads by combining pieces from standard benchmarks. We believe that our vision will spark new research avenues in database workload replay.

1 INTRODUCTION

Database benchmarking and workload replay have been extensively used to assess the performance of database engines and applications, guiding system and platform design decisions. They provide insights into performance characteristics and allow (a) developers to address performance bugs through improvements in the engine internals (e.g., in query processing, optimization, and storage) [1]; and (b) end users/database administrators (and more recently cloud providers) to fine-tune configurations [2, 3] or to identify and justify the optimal cloud service when migrating legacy customers [4]. Despite their widespread use, both approaches come with important drawbacks.

Database benchmarking. Standardized benchmarks, such as TPC-* and YCSB [5–8], are a common practice in the industry for measuring the performance of database systems. However, these benchmarks have under-performed on application-specific optimizations. With the rise of database-backed applications, database workloads are constantly evolving [9, 10] with different schemas and query types. A singleton standardized benchmark is not representative of the great variety of database workloads and cannot cover the full spectrum of analytic needs [11, 12].

Workload replay. On the other end, workload replay focuses directly on specific workloads, and thus is more amenable when a customized solution is required. It is able to reproduce performance inefficiencies and identify potential service disruptions more precisely for a particular workload. It helps diagnose the root cause of performance bugs and confirm the effectiveness of a fix or re-configuration. However, workload replay suffers from its own shortcomings.

Data accessibility. The most accurate approach for workload replay entails recording all user data and query history and replaying the exact same operations on the same data. Prior work on workload replay for data management systems requires full access to data and query history [13–19]. Unfortunately, accessing customer data and queries inevitably brings up privacy, security, and scalability issues, rendering these techniques inapplicable in many practical scenarios [4, 20]. Some recent works attempt to anonymize user data but still require full access to raw data [9].

Diverse hardware specifications. Existing workload replay techniques can be accurately used only on the same (or similar) hardware and software configurations as that of the original workload. This significantly limits the applicability of these approaches. It is very often the case that a given workload has to be replayed on a variety of settings: on-premise or on-cloud, centralized or distributed. Replaying the same workload on all these different configurations will result in very different performance results. Furthermore, it is almost impossible to simulate/access the exact same hardware settings as rigorous testing requires.

Our vision: learned workload synthesis. In this work, we introduce and make concrete steps to realize our *learned workload synthesis* vision: *given any profile (e.g., performance, or any other characteristics that can be measured quantitatively) corresponding to a workload that was run on a particular hardware/software configuration, synthesize a workload that has similar behavior when executed on the same (or any other) configuration.* Learned workload synthesis carefully combines ideas from both benchmarks and workload replay but overcomes their limitations. Using as input the performance profile of the original workload, we: (1) tailor our generated workload to the heterogeneous customer needs; (2) avoid accessing sensitive customer queries and data; and (3) target multiple hardware/software configurations for testing (details in Section 2).

Synthesizing a workload that follows a given performance profile on a specific type of hardware is notoriously hard, and an exhaustive approach is intractable. Inspired by construction toys (such as LEGO [21]), we propose Stitcher, a prototype that synthesizes SQL workloads by borrowing pieces of existing benchmarks that, when executed together, mimic the performance footprint of the original workload. To assist our search, we follow a learned approach that predicts the performance of a synthesized workload on any given hardware configuration. Our goal is for the synthetic workload to have the same characteristics (ideally in *all* measurable dimensions) as the original one, even if executed on

*Work done while interning at Microsoft.

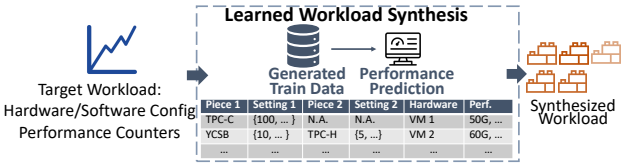

Figure 1: Learned workload synthesis overview.

different hardware/software. Our synthetically generated workloads can be used for performance benchmarking, configuration tuning and diagnostic purposes (details in Sections 3 and 4).

The preliminary results from our prototype of Stitcher are promising, suggesting that it is possible to synthesize workloads in this manner, as we show in Section 4. However, we are just getting started: we consider Stitcher to be the first step in realizing our learned workload synthesis vision. In Section 5, we describe technical challenges and several avenues for extending Stitcher. By sharing this framework, we hope it also encourages greater research in this area to help us bring this vision to light. Internally, the prototype of Stitcher has been used in two production application scenarios: (1) validating the SKU recommended for Azure migration workloads by replaying the respective synthesized workloads on various SKUs to evaluate the resource throttling [4] (the recommendation solution has been publicly released [22]); and (2) configuring Kubernetes [23] Pods that support SQL Server by replaying a selection of synthesized workloads with various characteristics (I/O intensive, large-scale data, high concurrency).

2 LEARNED WORKLOAD SYNTHESIS

In this section, we present our vision, also depicted in Figure 1. Learned workload synthesis takes as input the performance profile of the original workload, along with the hardware/software configuration this was executed on, and synthesizes a workload that has similar performance profile when executed on any other machine. Below we describe the inputs/outputs to the problem, the key ideas, and the challenges in building such a system.

2.1 Inputs and Outputs

Inputs. As opposed to traditional workload replay mechanisms that require full access to customer data and query history, our vision is to bypass this challenge by only requiring the following inputs: (1) performance counters of the original workload in the form of time series data (e.g., CPU, memory, IOPs, latency); and (2) the hardware/software configuration of the machine where these counters were collected. The performance trace collection is supported by many (low-overhead) profiling tools [24–27]. To further protect privacy, we only record aggregated performance data with configurable aggregation intervals, obfuscating timestamps and removing user IDs.

Outputs. The learned synthesis' output is a workload that "mimics" the customer workload in that it shares similar performance behavior when executed on any other machine. This synthetic workload can be generated by taking apart components of various standard benchmarks and re-composing parts of these SQL scripts such that its performance behavior follows the customer's performance footprint. We rely on mixing-and-matching the existing benchmark suites to generate new workloads. We make the assumption that most workloads can be captured by a particular combination of the massive available benchmark workloads (after tuning their configurations, e.g., scaling factor, query frequency).

2.2 Key Ideas

One of the most important building blocks of the synthesis process is the "learning", i.e., developing a comprehensive understanding of the performance profiles for different benchmark suites executed on different hardware/software configurations. If an exhaustive library of such observations exists, then, for any target workload, a searching process would be sufficient to enumerate and identify the best combination of the existing benchmarks that result in a performance profile that best matches that of the customer's real workload. As it is computationally intractable to execute and run all the combinations possible among the existing benchmark scripts, we turn to machine learning (ML) as a means to project, generalize and synthesize possible workloads given the limited observational (performance) data. That gives birth to our idea of ML-based learned workload synthesis.

For such a learning process, data is paramount in order to utilize the suite of ML tools available. More specifically, we need workloads and their respective performance traces collected from pre-specified hardware/software configurations. To build such a library, we can execute an array of workloads on a set of most dominant hardware/software configurations and record the subsequent performance profiles. This generated performance dataset "warm-starts" our prediction model, which we then rely on to interpolate the performance profile of any unseen combinations of synthetic workloads and hardware/software configurations. As more data is collected from our experiences with customers, we can continue to re-train our model to provide greater accuracy in generating new workloads that match the user's original performance footprint. To sum up, the synthesis comprises two main steps (see also Figure 1): (1) generation of training data to build libraries of performance profiles, and (2) workload performance prediction to support the synthesis of unseen customer workloads.

2.3 Challenges

The efficiency of our learned workload synthesis approach in creating new workloads that mimic the performance characteristics of the input workload is thus dictated by the ability to generate: (1) an extensive set of synthetic workload profiles, (2) an accurate predictor for performance profiles, and (3) an efficient algorithm to choose the optimal combination of basic benchmark pieces for a specific hardware/software configuration.

Workload profile generation. Our ability to mimic customer workload performance profiles with high fidelity depends on the comprehensiveness of our internal library of performance data that can be generated and used as the training data. We need an efficient solution to cover different types of workloads under a wide range of hardware/software configurations.

Performance prediction. It is infeasible to collect performance profiles for all possible combinations of workload and configurations. Therefore, we need to predict the performance footprint, utilizing generated profiles of synthetic workloads and the growing published performance history in literature. Different hardware and resource throttling are additional challenges.

Search algorithm. Finding the synthetic workload that shares the highest (predicted) profiling similarity as the customer workload is computationally intensive, as we need to search over an unlimited space of configuration settings, such as different benchmark type(s), scaling factors, etc. To improve the searching efficiency, a large number of advanced search algorithms can be

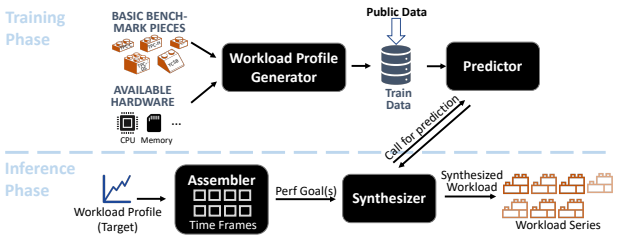

Figure 2: Stitcher architecture.

used, such as Bayesian Optimization [28], gradient descent [29], and graduated optimization [30].

3 STITCHER: REALIZING OUR VISION

In this section, we present Stitcher, the prototype we have built as a first step in realizing the learned workload synthesis vision.

3.1 System Architecture

The end-to-end synthetic workload generation process of Stitcher consists of a *training phase* and an *inference phase* (see Figure 2).

The *training phase* delineates the performance behavior for each synthetic workload. For example, it helps us understand what types of workloads are memory intensive or how to tune our workload mixture to increase/decrease IO latency. In this phase, we develop a *Workload profile generator* that executes an extensive library of workloads (constructed by basic benchmark pieces) and collects their performance profiles on various hardware/software configurations. This training phase aims at generating the data we need to build the *Predictor* that we ultimately rely on to predict the performance profile given any workload types and hardware/software configurations.

In the *inference phase*, based on the input performance counters of the customer workload (our target), an *Assembler* extracts representative time-frames from the full time series data of workload traces and constructs performance goal(s). It then invokes the *Synthesizer*, which leverages the pre-trained predictor (from the training phase) to search for the proper mixture of standard benchmark pieces to reconstruct a synthetic workload that, when executed, resembles the target performance footprint.

3.2 Basic System Components

3.2.1 Basic benchmark pieces. We collect an initial set of basic pieces from open-source database benchmarks, including TPC-C [7], TPC-DS [6], TPC-H [5], and YCSB [31] with different database sizes (i.e., scaling factors) that are available through the OLTP-Bench [32]. We adopt this benchmark set for our prototype, as they cover a large range of database sizes, data schemata, and query schemata. Stitcher framework could be easily extended to incorporate additional basic benchmark pieces. Basic benchmark pieces have two configurations: (1) query frequency (**Freq**uency): the upper limit of submitted queries from one client per second (e.g., for TPC-C and YCSB) or per minute (e.g., for TPC-H and TPC-DS); and (2) concurrent clients (**Con**currency): the number of active clients that submit queries continuously.

3.2.2 Workload profile generator. It conducts experiments and collects performance profiles as training data for the *Predictor*, focusing on the configuration subspace where performance is significantly impacted by workload **Freq**uency and **Con**currency. We use a joint enumeration and random sampling technique to collect training data by varying the combination of basic workload pieces and their respective frequency and concurrency. We execute each workload mixture for a period of time (e.g., 5 minutes) and collect performance counters on servers on a range of Azure machines. For synthetic workloads that are generated only from one basic piece from one benchmark, we test more frequency and concurrency combinations. For synthetic workloads that are generated by multiple basic pieces, we randomly sample a fraction (e.g., 1/30) of all the possible (enumerated) combinations. Our library could be extended to existing literature by reporting their profiling results in the training data. Stitcher could achieve better performance if informed of the impact of database scaling factor and query concurrency/frequency on its performance behavior to further filter candidates.

3.2.3 Predictor. The *Predictor* is a global model that takes hardware/software configurations and workload settings, and outputs performance prediction.

Performance counters. We believe that if the performance footprint of the synthetic workload matches that of the target (on all measurable dimensions), it should then follow that the two workloads share similar characteristics. In this prototype, we focus on four dimensions of performance that the Microsoft Data Migration Assistant (DMA) [22] collects: vCore usage (CPU cores used by SQL server); memory usage (memory used by SQL server); dataIOPs (IO operations SQL server processed per second); and latency[1] (end-to-end latency of one IO operation). Note that Stitcher is extensible to any performance dimension. Future work will include more (e.g., wait stats) that can help in scenarios that require customer workloads to be executed on different hardware/software such as migration (see Section 5), or even other query characteristics (such as physical plans).

ML prediction model. Workload performance is impacted by multiple aspects, including (1) the workload itself: the basic benchmark pieces and their configuration; and (2) the system where the workload executes (e.g., hardware specifications and software versions). Their impacts are extremely hard to manually model in a quantitative way, as they interfere with each other and may cause unpredictable compatibility problems. Fortunately, with the sufficient data collected by workload profile generator, we are able to use ML techniques to capture their relationship.

Our prediction model was trained on the time series performance data collected with our generator for different types of synthesized workloads in different environments. It takes workload type, hardware-software combination, **Freq** and **Con** of each basic piece as input, and outputs performance counters.

For our prototype, we started with linear models to predict workload performance, taking into consideration that (1) we tend to observe a linear relationship between performance and workload given the same (or similar) hardware/software (Section 4.2); (2) simpler models perform well with limited training data; and (3) linear models are naturally more explainable. We built separate prediction models for each workload type and hardware/software specification, mainly based on vCore number. Each model is built upon performance counter data collected on a limited range of hardware/software settings. It takes **Freq**, **Con**, and **Freq**×**Con** as independent variables, and predicts *vCore usage*, *memory usage*, *ln(DataIOPs)*, and *ln(latency)*. Future work will include improving and generalizing our predictors to untested hardware/software settings (Section 5).

3.2.4 Synthesizer. Given a customer's performance footprint (the target), the *Synthesizer* finds a workload that has the closest

[1] It is not the inverse of data IOPs, as SQL server may have idle time between queries.

ID	vCPU	Memory	Cache	Throughput	Disk
D32s_v4	32 cores	128 GB	800 GB	308000 IOPs	2TB SSD
D16as_v4	16 cores	64 GB	400 GB	154000 IOPs	
D8s_v3	8 cores	32 GB	200 GB	12000 IOPs	
D4s_v3	4 cores	16 GB	100 GB	6000 IOPs	

Table 1: Hardware platforms used in our experiments (ID refers to machine type provided by Azure Cloud)

performance match as estimated by the *Predictor*. After identifying the workload types that have the most similar performance, it applies Bayesian optimization [28] on each workload type's candidates to find the optimal **Freq** and **Con** settings by minimizing the mean squared percentage error (MSPE) between the estimated performance and target. The *Synthesizer* returns the top-n candidates with the smallest error. Depending on the application, the Synthesizer can also put different weights on different performance profiling dimensions.

3.2.5 Assembler. The *Assembler* receives the time series performance data for the customer workload, extracts representative time-frames and passes those as the target for the *Synthesizer*. After the *Synthesizer* returns the optimal workload configurations for each input time-frame, the *Assembler* replays those synthetic workloads sequentially if required. In this prototype, we replay the whole time series of the input performance counters when at least one dimension of the resource usage is non-zero. Following this conservative policy, we retain the majority of the time series data and thus generate workloads that mimic the behavior for the entire duration of the input performance footprint. One important future direction is to improve on this selection algorithm to reduce the representative time periods and further increase its efficiency. Oftentimes, we find that a significant fraction of the performance footprint is non-critical and consists of relatively low resource utilization, and thus does not need to be considered.

3.3 Implementation

The Stitcher framework is designed to be generalizable and agnostic to various DBMS, platforms, and DB workloads. In preliminary experiments, we implemented a prototype on a subset of them:

Platform. The prototype is implemented for Microsoft SQL server 2019, which we installed in a range of Azure VM types [33] (to simulate different hardware/software configurations).

Basic benchmark pieces. We use the OLTP-Bench framework [32] to construct basic benchmark pieces, taking workload type, scale, query frequency, and number of clients as input. We adopt OLTP-Bench as it is an extensible database benchmarking framework that can easily incorporate new types of database benchmark.

Performance counters. It leverages the DMA tool [22] to collect performance counters for CPU, memory, IOPs, and latency, as described in Section 3.2.3 (more counters can be added), which is lightweight, has little overhead and can be easily installed.

Algorithm. Our prototype is implemented in Python. The *Predictor* uses scikit-learn [34] to train linear regression models. The *Synthesizer* adopts the Bayesian optimization package [35] to improve the search efficiency, using conservatively 100 iterations and returning top $n = 3$ candidates for comparison.

4 PRELIMINARY RESULTS

In this section, we evaluate our prototype of Stitcher and present various insights from our experiments. We release our experimental data and results online [36].

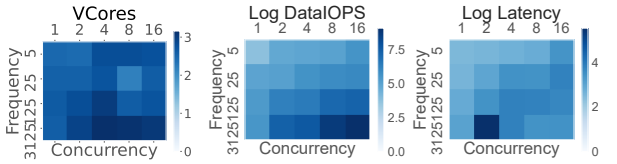

Figure 3: TPC-C performance correlation with workload frequency and concurrency (larger value has darker color)

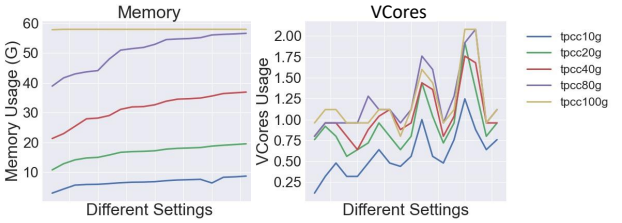

Figure 4: Memory/VCore usage of TPC-C under different database sizes on *D16as_v4* after restarting

4.1 Experimental setup

4.1.1 Hardware platform. As the 4vCore, 8vCore and 16vCore Azure General Purpose (GP) and Managed Instance (MI) [37], offerings account for 98% of the total demand, we use the Azure VMs [33] highlighted in Table 1 to provide sufficient coverage of realistic customer hardware platforms. We use *D8s_v3* instances for the client (to submit queries) and all platforms for running SQL Server to collect training data for the *Predictor* respectively.

4.1.2 Benchmark pieces. We use the following 6 benchmark pieces: TPC-C with 10G/100G data, YCSB with 100G data, TPC-H with 100G/300G data, and TPC-DS with 100G data. Using these pieces, we: (1) generate workloads to train the Predictor, and (2) generate workloads to evaluate the Synthesizer.

4.2 Performance Correlations

To gain insights on performance bottlenecks, we run TPC-C (100G data) with different settings. Figure 3 shows its performance (color) under different **Freq** (y-axis) and **Con** (x-axis) values. Despite the significant noise arising from variations in network connections and runtime, we observe a clear trend: *vCore usage*, *ln(DataIOPs)*, and *ln(Latency)* have an approximate linear relationship with **Freq** and **Con**[2]. Hence, we can use linear regression models to capture the dynamics between resource usage (prediction targets) and workload frequency and concurrency (inputs) per hardware configuration.

Memory usage is a special case. Our training data includes very limited low-memory data points, as SQL server tends not to release memory even when the workload is light. Further experimentation shows that the actual memory usage (collected after restarting the server each time when configurations change) is correlated to the scale factor and settings of benchmark pieces. Figure 4 demonstrates the actual memory/vCore usage of TPC-C benchmarks under different database sizes on Azure *D16as_v4* machine where on the x-axis, **Freq** is increasing faster than **Con** (e.g., $\{c = 1, f = 1\}, \{c = 1, f = 2\}, \ldots, \{c = 2, f = 1\}$). We observe that the upper limit of memory consumption is closer to the smaller value of database size and memory capacity. Therefore, more accurate counters to capture the true memory usage can dramatically improve the prediction even with a simple model.

[2]Note that by default SQL Server tends to not release memory once it has acquired it, so memory usage remains close to machine capacity.

	vCore	Memory (G)	*ln*(DataIOPs)	*ln*(Latency) (ms)
MAPE	18%	3%	13%	12%
MAE	1.69	1.23	0.66	0.62
MAE'	1.47	0.63	0.54	0.69

Table 2: Average error of predictor for *D16as_v4*

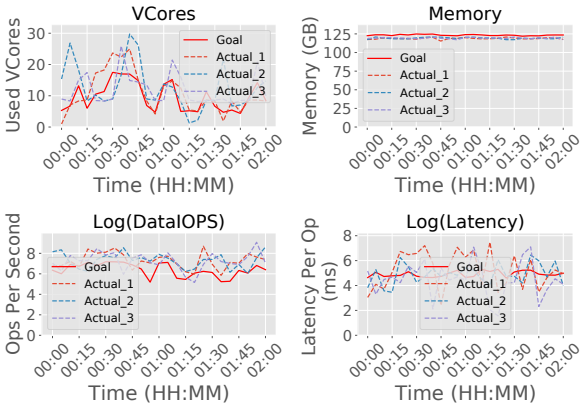

Figure 5: Actual performance of Synthesized workloads on *D32s_v4* mimicking real customer workload traces.

Insight: There exists an approximate linear relationship between benchmark configurations and performance counters. For memory, there exists a direct mapping from the database sizes (i.e., scaling factors), and the **Freq** and **Con** values to memory usage.

4.3 Predictor Results

To train the predictor, our workload profile generator collected >1,500 training data, over 250 CPU hours, covering 31 different workload type combinations.

Table 2 shows the mean absolute percentage error (MAPE), mean absolute error (MAE), and median absolute error (MAE') for the Predictor on *D16as_v4*. Overall, our prototype achieves 3-18% error on predicting performance.Prediction error for *vCore usage* is relatively low, partly because vCore usage is less sensitive to runtime variations. *DataIOPs* and *Latency* have larger errors for two reasons: (1) error increases when scaling back from their logarithmic value in the linear regression model; (2) runtime variation introduces noise to the training data.

Insight: Some resources are more challenging to predict than others. While our trained linear models show low prediction error for some resources, they perform poorly in accurately capturing others.

4.4 Synthesizer Results

4.4.1 Evaluate with real customer's workload. We first evaluated our synthesizer with a real Azure customer's workload. Stitcher takes customer's performance series as its target, and synthesizes three workload series, referring to the top-3 candidates obtained by the *Synthesizer*.

Figure 5 shows the target performance, and the actual performance achieved by synthesized workload on *D32s_v4* machine. The result shows that Stitcher is able to replay real customer's workload with the mean absolute percentage errors (MAPEs) for CPU, Memory, Log(DataIOPS), Log(Latency) as 33%, 4%, 17%, and 23% respectively for the best candidate.

	ID	VCore	Mem.	*ln*(IOPs)	*ln*(Lat.)
Same Machine	D16as_v4	27%	2%	23%	18%
Stronger Machine	D32s_v4	52%	16%	95%	64%
Weaker Machine	D8s_v3	48%	5%	10%	15%
	D4s v3	60%	5%	14%	14%

Table 3: Average error of synthesizer for *D16as_v4* mimicking synthetic workload

These results show that Stitcher is able to synthesize new workloads that, when replayed, exhibit relatively similar performance behavior as that of the customer's performance footprint on a similar (or less powerful) machine.

4.4.2 Evaluate with synthetic workload. We further evaluated Stitcher with 82 workloads that we randomly generated with the basic benchmark pieces to capture a large variety of different workload characteristics. We (1) execute the **Target** workloads on *D16as_v4* for 5 minutes to obtain the performance target, (2) invoke Stitcher using these targets, collect the synthesized workloads output by Stitcher (**Synthesized**), and (3) compare the performance differences between **Target** and **Synthesized** workload on another *D16as_v4* machine.

As shown in the first row of Table 3, Stitcher has 2-27% average error of performance counters when running the **Synthesized** workload on the same machine that **Target** was originally run. In terms of the time dimension, the performance time series of the **Synthesized** workload is similar to that of the **Target** workload.

Insight: Stitcher is able to synthesize new workloads based on real customer performance profiles that exhibit relatively similar behavior on similar (or less powerful) Azure machines.

4.5 Challenges in Workload Migration

As one of the use cases of Stitcher is generating synthetic workloads to support migration from on-prem workloads to the cloud, or between cloud vendors, we tested the ability of our system in synthesizing workloads that, not only perform similarly as that of the original workload on the same hardware/software setting, but also perform similarly on platforms the customer may want to migrate their workloads to. We thus extended the experiments in Section 4.4.2 and executed the **Target** and **Synthesized** workload on all machines in Table 1, other than *D16as_v4*.

As shown in Table 3, Stitcher has 48-60% error on *vCore usage*. For the other performance counters, Stitcher achieves 5-15% accuracy on less powerful machines (*D8s_v3*, *D4s_v3*), which is similar to that of the original machine (*D16as_v4*). However, Stitcher has much higher error on a more powerful machine (*D32s_v4*).

This increased error is caused by the different performance bottleneck of various hardwares. Time series performance data do not fully capture the bottlenecks in the execution of customer workloads. As a result, inaccuracies arise when replaying workloads on machines that have higher resource capacity. In our future work, the predictor would also feauturize hardware, which can help Stitcher assemble synthetic workloads that mimic the customers' with higher fidelity across hardware.

5 THE ROAD AHEAD

In this paper, we demonstrate the ability of Stitcher to mix-and-match basic benchmark pieces, such as TPC-C and TPC-DS, to synthesize a wide range of workloads that, when executed on similar hardware/software, mimic the customers' performance footprint with relatively good accuracy. Yet, much work remains to improve Stitcher generalizability and predictor accuracy.

Predictor: Unseen Hardware. Our *Predictor* prototype does not completely resolve the mapping challenge between (basic pieces, hardware/software) to performance counters. Stitcher, in its current state, does not generalize to synthesize workloads on unseen hardware/software combinations outside the training set. As introduced in Section 3.2.3, our prototype focus on a limited set of popular Azure cloud machines. It trains a *hardware-specific* Predictor model with training data uniquely collected from each of these machines.

To address this limitation, we first trained models that clustered machines with similar configurations and utilized these groups for prediction. Such clustering did not significantly improve our ability to accurately predict how a workload would perform on a different hardware/software configuration, nor did it generalize well across clusters. Additional work remains in developing a *global model* that predicts the performance under all Azure cloud machines or under competing cloud hardware/software configurations. For such a model, the hardware/ software settings that the synthesized workload will be tested need to become *inputs/features* to the predictor.

Our final goal is to build a general model that assesses multiple aspects simultaneously. It includes building a predictor that assesses how the customer's original workload will perform on new hardware and leveraging our synthesizer to generate workloads that mimic this predicted performance. Lack of extensive (performance) data on how our set of synthesized workloads behave when previous throttling was resolved on larger capacity hardware remains the primary hurdle towards developing such a global model. However, we expect the inclusion of additional performance counters, such as wait stats and other dimensions that capture resource throttling, will significantly improve the performance of our Synthesizer, particularly for the predictability under different hardware/software. These additional performance constraints would effectively coerce our model to identify a better combination of our benchmark pieces that results in a performance profile that better matches that of the customer.

Predictor: Database Configurations. Database workload performance is affected, not only by the hardware specifications, but also by the database configurations. In our prototype, Stitcher assumes that customers use default settings, but, in reality, customers often tune configurations for better performance. Future work will include addressing database performance-related configurations in the Predictor model, e.g., buffer pool size and blocked process threshold.

Predictor: Memory Consumption. Memory consumption is related to database size, machine capacity, and execution history. To further understand this impact, we test TPC-C benchmarks under different database sizes, as well as, **Freq** & **Con** settings. As some DBMS, including MS SQL server, tend not to release memory, a light workload would still exhibit high memory usage if it is executed after a heavy workload. To resolve this transition, we restart the machine to refresh memory. Future work entails investigating how to better measure memory and how to autogenerate a more diverse set of database sizes to achieve particular memory usage patterns.

Assembler: Warm Up Stage. Stitcher concatenates different workloads in the time dimension neglecting the effect of database warm-up. While the warm-up stage has a small effect with a longer sampling window and data aggregation, it is important to consider its impact on the synthetic workload's footprint. This would allow us to generate synthetic workloads with higher fidelity to that of the customer. Future work includes taking this warm-up effect into consideration in our synthesis algorithm.

Synthesizer: Feedback Loop. In our current design, *Synthesizer* generates workloads without validation. The workload performance should be validated by replaying the synthetic workload on the machine where the target performance footprint was collected (if possible). Future work thus entails developing a feedback loop in which we consider new workloads that minimize the performance difference between that of the (replayed) synthetic workload and the target footprint.

Extend Stitcher to Various Systems. Workload replay is an important and challenging problem for performance-sensitive systems, including distributed systems, micro-services, and network systems. While our current prototype is designed for single-node database applications, we believe that our vision of learned workload synthesis can be beneficial to other system domains.

6 RELATED WORK

Previous work [5–7, 31, 38–42] has focused on designing benchmarks for DBMS to capture different types of application scenarios, e.g., transaction processing [7], decision support [5, 6], and cloud serving [31]. While they cover a number of scenarios, real-world production workload might introduce even greater diversity, which is difficult to capture with a single benchmark.

To resolve this issue, prior work proposes workload replay for realistic reproduction of workload history. They trace sessions to replay the entire workload and synchronization schemes at a cost of considerable overhead [13–16], or use workload compression to find a representative subset [9, 17–19]. DIAMetrics [9] is one example that scrambles customer data and query history, anonymizes specific components, and extracts a representative workload. While anonymized, it still requires data access to the customer's raw data. Other studies implement capture-and-replay at the network [43] and OS-kernel [44] levels. Another line of work [9, 17–19] focuses on workload compression, using a representative subset to characterize the original one. Unlike Stitcher, these works require full access to user data or query history.

There also exist solutions [45–48] that focus on synthesizing testing workloads for database applications. They generate test cases and find bugs at the application level, neglecting potential issues at the DBMS level.

Another line of work optimizes database [49–52] and data structures [53, 54] automatically with ML techniques, predicting query performance from specific database configurations. A recent work [55] also points out the principle of designing benchmarks for ML-optimized databases. These works target different problems from Stitcher.

7 CONCLUSION

Workload generation is an important means of understanding and diagnosing system performance, which underpins platform and application development. In this paper, we propose Stitcher, a flexible framework for synthesizing SQL workloads that, when executed, mimic the performance footprint of real workloads. Stitcher synthesizes new workloads by leveraging a comprehensive library of basic workload scripts derived from standard benchmarks. Synthesized workloads are demonstrated to achieve close-to-history estimated performance on a variety of hardware/-software settings. Preliminary results show the feasibility of this methodology and suggest several avenues for future improvement.

REFERENCES

[1] J. Yang, C. Yan, P. Subramaniam, S. Lu, and A. Cheung, "How not to structure your database-backed web applications: a study of performance bugs in the wild," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pp. 800–810, IEEE, 2018.

[2] S. Agrawal, S. Chaudhuri, L. Kollar, A. Marathe, V. Narasayya, and M. Syamala, "Database tuning advisor for microsoft sql server 2005," in *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pp. 930–932, 2005.

[3] G. Li, X. Zhou, S. Li, and B. Gao, "Qtune: A query-aware database tuning system with deep reinforcement learning," *Proceedings of the VLDB Endowment*, vol. 12, no. 12, pp. 2118–2130, 2019.

[4] J. Cahoon, W. Wang, Y. Zhu, K. Lin, S. Liu, R. Truong, N. Singh, C. Wan, A. Ciortea, S. Narasimhan, and S. Krishnan, "Doppler: A framework for automated sku recommendation in migrating sql workloads to the cloud," *PVLDB*, vol. 15, no. 12, pp. 3509–3521, 2022.

[5] T. Benchmarks, "Tpc-h." Online document, http://tpc.org/tpch, 2021.

[6] T. Benchmarks, "Tpc-ds." Online document, http://tpc.org/tpcds, 2021.

[7] T. Benchmarks, "Tpc-c." Online document, http://tpc.org/tpcc, 2021.

[8] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM symposium on Cloud computing*, pp. 143–154, 2010.

[9] S. Deep, A. Gruenheid, K. Nagaraj, H. Naito, J. Naughton, and S. Viglas, "Diametrics: benchmarking query engines at scale," *ACM SIGMOD Record*, vol. 50, no. 1, pp. 24–31, 2021.

[10] Y. Zhu, M. Interlandi, A. Roy, K. Das, H. Patel, M. Bag, H. Sharma, and A. Jindal, "Phoebe: A learning-based checkpoint optimizer," *PVLDB*, vol. 14, no. 11, pp. 2505–2518, 2021.

[11] H. R. Taheri, "Does the tpc still have relevance?," *19th International Workshop on High Performance Transaction Systems (HPTS)*, 2017.

[12] J. Jacobs, "Snowflake vs databricks: Tpcs-ds benchmark wars – who cares?." Online document, https://www.linkedin.com/pulse/snowflake-vs-databricks-tpcs-ds-benchmark-wars-who-cares-jacobs/?trk=public_post-content_share-article, 2021.

[13] L. Galanis, S. Buranawatanachoke, R. Colle, B. Dageville, K. Dias, J. Klein, S. Papadomanolakis, L. L. Tan, V. Venkataramani, Y. Wang, *et al.*, "Oracle database replay," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pp. 1159–1170, 2008.

[14] K. Yagoub, P. Belknap, B. Dageville, K. Dias, S. Joshi, and H. Yu, "Oracle's sql performance analyzer.," *IEEE Data Eng. Bull.*, vol. 31, no. 1, pp. 51–58, 2008.

[15] Microsoft, "Overview of database experimentation assistant." Online document, https://docs.microsoft.com/en-us/sql/dea, 2021.

[16] K. Morfonios, R. Colle, L. Galanis, S. Buranawatanachoke, B. Dageville, K. Dias, and Y. Wang, "Consistent synchronization schemes for workload replay," *Proceedings of the VLDB Endowment*, vol. 4, no. 12, pp. 1225–1236, 2011.

[17] S. Chaudhuri and V. R. Narasayya, "An efficient, cost-driven index selection tool for microsoft sql server," in *VLDB*, vol. 97, pp. 146–155, Citeseer, 1997.

[18] S. Chaudhuri, A. K. Gupta, and V. Narasayya, "Compressing sql workloads," in *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pp. 488–499, 2002.

[19] S. Y. Philip, H.-U. Heiss, S. Lee, and M.-S. Chen, "On workload characterization of relational database environments.," *IEEE Trans. Software Eng.*, vol. 18, no. 4, pp. 347–355, 1992.

[20] A. Miranskyy, A. Hamou-Lhadj, E. Cialini, and A. Larsson, "Operational-log analysis for big data systems: Challenges and solutions," *IEEE Software*, vol. 33, no. 2, pp. 52–59, 2016.

[21] T. L. Group, "Lego." Online document, https://www.lego.com/, 2021.

[22] Microsoft, "Overview of data migration assistant." Online document, https://docs.microsoft.com/en-us/sql/dma, 2021.

[23] C. N. C. Foundation, "Kubernetes: Production-grade container orchestration." Online document, https://kubernetes.io/, 2022.

[24] A. Pesterev, N. Zeldovich, and R. T. Morris, "Locating cache performance bottlenecks using data profiling," in *Proceedings of the 5th European conference on Computer systems*, pp. 335–348, 2010.

[25] R. Chard, K. Chard, B. Ng, K. Bubendorfer, A. Rodriguez, R. Madduri, and I. Foster, "An automated tool profiling service for the cloud," in *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pp. 223–232, IEEE, 2016.

[26] A. V. Do, J. Chen, C. Wang, Y. C. Lee, A. Y. Zomaya, and B. B. Zhou, "Profiling applications for virtual machine placement in clouds," in *2011 IEEE 4th international conference on cloud computing*, pp. 660–667, IEEE, 2011.

[27] J. Zhang and R. J. Figueiredo, "Application classification through monitoring and learning of resource consumption patterns," in *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*, pp. 10–pp, IEEE, 2006.

[28] M. Pelikan, D. E. Goldberg, E. Cantú-Paz, *et al.*, "Boa: The bayesian optimization algorithm," in *Proceedings of the genetic and evolutionary computation conference GECCO-99*, vol. 1, pp. 525–532, Citeseer, 1999.

[29] S. Ruder, "An overview of gradient descent optimization algorithms," *arXiv preprint arXiv:1609.04747*, 2016.

[30] E. Hazan, K. Y. Levy, and S. Shalev-Shwartz, "On graduated optimization for stochastic non-convex problems," in *International conference on machine learning*, pp. 1833–1841, PMLR, 2016.

[31] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM symposium on Cloud computing*, pp. 143–154, 2010.

[32] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudré-Mauroux, "Oltp-bench: An extensible testbed for benchmarking relational databases," *PVLDB*, vol. 7, no. 4, pp. 277–288, 2013.

[33] Microsoft, "Azure virtual machines." Online document, https://azure.microsoft.com/en-us/free/virtual-machines/, 2021.

[34] scikit learn, "scikit-learn: machine learning in python." Online document, https://scikit-learn.org/stable/, 2021.

[35] F. Nogueira, "Bayesian Optimization: Open source constrained global optimization tool for Python." Online document, https://github.com/fmfn/BayesianOptimization, 2014.

[36] Microsoft, "Stitcher: Learned workload synthesis from historical performance footprints, data release." Online document, https://publicstitcher.blob.core.windows.net/stitcher/stitcher_data.zip, 2022.

[37] Microsoft, "Azure sql managed instance.," 2021. https://azure.microsoft.com/en-us/products/azure-sql/managed-instance/.

[38] D. Bitton, D. J. DeWitt, and C. Turbyfill, "Benchmarking database systems-a systematic approach," tech. rep., University of Wisconsin-Madison Department of Computer Sciences, 1983.

[39] P. Boncz, A.-C. Anatiotis, and S. Kläbe, "Jcc-h: adding join crossing correlations with skew to tpc-h," in *Technology Conference on Performance Evaluation and Benchmarking*, pp. 103–119, Springer, 2017.

[40] M. J. Carey, D. J. DeWitt, and J. F. Naughton, "The 007 benchmark," *ACM SIGMOD Record*, vol. 22, no. 2, pp. 12–21, 1993.

[41] M. J. Carey, D. J. DeWitt, J. F. Naughton, M. Asgarian, P. Brown, J. E. Gehrke, and D. N. Shah, "The bucky object-relational benchmark," in *Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, pp. 135–146, 1997.

[42] A. Dey, A. Fekete, R. Nambiar, and U. Röhm, "Ycsb+ t: Benchmarking web-scale transactional databases," in *2014 IEEE 30th International Conference on Data Engineering Workshops*, pp. 223–230, IEEE, 2014.

[43] E. solutions, "ireplay: Database workload capture and replay." Online document, https://www.exact-solutions.com/products/ireplay, 2021.

[44] K. Kim, C. Lee, J. H. Jung, and W. W. Ro, "Workload synthesis: Generating benchmark workloads from statistical execution profile," in *2014 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 120–129, IEEE, 2014.

[45] C. Binnig, D. Kossmann, E. Lo, and M. T. Özsu, "Qagen: generating query-aware test databases," in *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pp. 341–352, 2007.

[46] M. Emmi, R. Majumdar, and K. Sen, "Dynamic test input generation for database applications," in *Proceedings of the 2007 international symposium on Software testing and analysis*, pp. 151–162, 2007.

[47] K. Pan, X. Wu, and T. Xie, "Guided test generation for database applications via synthesized database interactions," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 23, no. 2, pp. 1–27, 2014.

[48] K. Pan, X. Wu, and T. Xie, "Automatic test generation for mutation testing on database applications," in *2013 8th International Workshop on Automation of Software Test (AST)*, pp. 111–117, IEEE, 2013.

[49] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, "The case for learned index structures," in *Proceedings of the 2018 international conference on management of data*, pp. 489–504, 2018.

[50] R. Marcus, P. Negi, H. Mao, C. Zhang, M. Alizadeh, T. Kraska, O. Papaemmanouil, and N. Tatbul, "Neo: A learned query optimizer," *arXiv preprint arXiv:1904.03711*, 2019.

[51] P. Negi, R. Marcus, H. Mao, N. Tatbul, T. Kraska, and M. Alizadeh, "Cost-guided cardinality estimation: Focus where it matters," in *2020 IEEE 36th International Conference on Data Engineering Workshops (ICDEW)*, pp. 154–157, IEEE, 2020.

[52] P. Negi, R. Marcus, A. Kipf, H. Mao, N. Tatbul, T. Kraska, and M. Alizadeh, "Flow-loss: learning cardinality estimates that matter," in *VLDB*, 2021.

[53] S. Idreos, K. Zoumpatianos, S. Chatterjee, W. Qin, A. Wasay, B. Hentschel, M. Kester, N. Dayan, D. Guo, M. Kang, *et al.*, "Learning data structure alchemy," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 42, no. 2, 2019.

[54] S. Idreos and T. Kraska, "From auto-tuning one size fits all to self-designed and learned data-intensive systems," in *Proceedings of the 2019 International Conference on Management of Data*, pp. 2054–2059, 2019.

[55] L. Bindschaedler, A. Kipf, T. Kraska, R. Marcus, and U. F. Minhas, "Towards a benchmark for learned systems," in *2021 IEEE 37th International Conference on Data Engineering Workshops (ICDEW)*, pp. 127–133, IEEE, 2021.