Appears in the Proceedings of the 55th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2022

Datamime: Generating Representative Benchmarks by Automatically Synthesizing Datasets

Hyun Ryong Lee Daniel Sanchez
Massachusetts Institute of Technology
{*hrlee, sanchez*}*@csail.mit.edu*

***Abstract*—Benchmarks that closely match the behavior of production workloads are crucial to design and provision computer systems. However, current approaches fall short: First, open-source benchmarks use public datasets that cause different behavior from production workloads. Second, black-box workload cloning techniques generate synthetic code that imitates the target workload, but the resulting program fails to capture most workload characteristics, such as microarchitectural bottlenecks or time-varying behavior.**

Generating code that mimics a complex application is an extremely hard problem. Instead, we propose a different and easier approach to benchmark synthesis. Our key insight is that, for many production workloads, the program is publicly available or there is a reasonably similar open-source program. In this case, generating the right *dataset* is sufficient to produce an accurate benchmark.

Based on this observation, we present *Datamime*, a profile-guided approach to generate representative benchmarks for production workloads. Datamime uses the performance profiles of a target workload to generate a dataset that, when used by a benchmark program, behaves very similarly to the target workload in terms of its microarchitectural characteristics.

We evaluate Datamime on several datacenter workloads. Datamime generates synthetic benchmarks that closely match the microarchitectural features of these workloads, with a mean absolute percentage error of 3.2% on IPC. Microarchitectural behavior stays close across processor types. Finally, time-varying behaviors are also replicated, making these benchmarks useful to e.g. characterize and optimize tail latency.

***Keywords*-benchmarking; workload generation.**

I. Introduction

Representative benchmarks are critical to computer architects and systems designers. Benchmarks allow architects to design hardware tailored to their target applications. This is especially hard to do for modern datacenter workloads [12], which often operate on confidential and proprietary data.

Unfortunately, existing datacenter benchmarks are rarely representative of production workloads. These benchmarks use synthetic or publicly available datasets that are very different from those in production [12, 31], or use traces that are representative at the time of construction but become outdated as user data changes over time without continued maintenance [16, 19, 54]. Fig. 1(left) shows an example of this problem: it reports the instructions per cycle (IPC) on a Broadwell processor for `memcached` running with two datasets: one with a publicly available dataset representative of Facebook's production environment [3] (in blue), and another with the default dataset used in the Tailbench benchmark suite [31] (in red). IPC is 2.4× lower for the Tailbench dataset. This stark difference comes from very different microarchitectural behavior. For example, Fig. 1(center) shows that the Tailbench dataset incurs 3.2× higher instruction cache misses per kilo-instruction (ICache MPKI), but other metrics (e.g., branch mispredictions and data misses) are also very different.

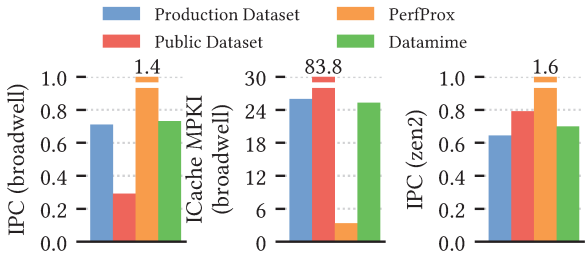

Figure 1: Accuracy comparison of benchmark generators when mimicking `memcached` with a Facebook production dataset: IPC (left) and ICache MPKI (center) on an Intel Broadwell CPU, and IPC (right) on an AMD Zen 2 CPU. `memcached` with a public dataset (TailBench's default) and the PerfProx-generated code are very different from the production workload. By contrast, Datamime produces a dataset that makes `memcached` closely mimic the production workload.

Alternatively, black-box workload cloning techniques [4, 5, 27, 41] generate synthetic *code* that mimics the behavior of a target workload. These techniques directly analyze the production workload to generate code, so in theory they need not suffer from the mismatch between production and public datasets. But in practice, trying to mach the behavior of a complex program with synthetic code is very complicated. Thus, the produced benchmark does not preserve the structure and high-level behavior of the production workload, resulting in an inaccurate benchmark. Fig. 1 shows this problem: when PerfProx [41], a state-of-the-art workload cloning technique, is used to clone `memcached`, the resulting benchmark has a 1.94× higher IPC on Broadwell. PerfProx also produces very different microarchitectural behavior, with 7.76× lower ICache MPKI. Beyond this mismatch, the resulting clone does not capture the overall structure of the workload, e.g., its request-driven nature and time-varying behavior, which is crucial to study and optimize many key metrics, like tail latency, and mechanisms, like OS interactions. For these

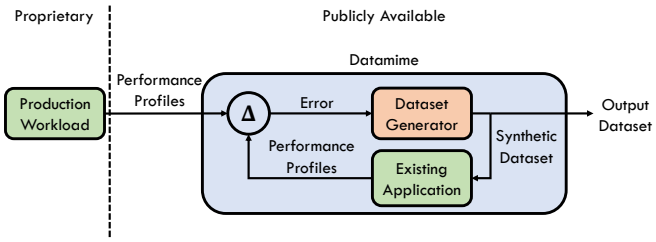

Figure 2: Brief overview of Datamime. Datamime uses the difference in performance profiles between the synthetic benchmark and the production workload to produce representative synthetic datasets.

reasons, black-box workload cloning has seen little adoption within the architecture community.

We introduce *data-centric benchmark generation*, a new insight to generate representative benchmarks. We observe that the code used by production workloads is often publicly available (e.g., `memcached`), or a reasonably similar open-source alternative exists (e.g., an open-source database or search engine). Thus, instead of trying to generate code as in prior work, it is simpler and more effective to generate a representative *dataset*, i.e., one that makes the public code closely mimic the target workload.

Based on this observation, we present *Datamime*, a technique to generate representative benchmarks from production workloads by synthesizing datasets. Fig. 2 gives an overview of Datamime. Datamime uses three inputs: *(1)* performance profiles from the target workload, *(2)* an existing program, and *(3)* a dataset generator for the program. Datamime then searches for a dataset that produces the closest performance profiles to those of the target workload.

Fig. 1 shows that, unlike prior work, Datamime accurately mimics the production Facebook workload. IPC and ICache MPKI are within 2.8% and 2.6% of the target workload, and results carry over to other microarchitectures: IPC on an AMD Zen 2 machine (right of Fig. 1) is within 8.5%, whereas other techniques vary significantly. Because our performance profiles include many microarchitectural metrics and time-varying behavior, Datamime accurately captures these aspects as well, as we will see later.

We evaluate Datamime in a single-node setup across five datacenter workloads. Datamime is able to synthesize datasets that differ from the target workload by a mean absolute percentage error of 3.2% on IPC. In comparison, state-of-the-art black-box cloning suffers a 42.9% mean absolute percentage error on IPC. Trends are similar on other microarchitectural metrics, like memory traffic.

In addition, Datamime closely matches the *distributions* of performance counters and CPU utilization. Datamime can prioritize matching certain metrics to meet the benchmark designer's needs. And Datamime's optimizer is fast, requiring 6–13 hours of serial work to produce an accurate benchmark.

Overall, Datamime makes it easy to produce benchmarks that are representative of production services. Datamime requires collecting a one-time, low-overhead profile of the target workload. While this profiling step needs to be done by the operator of the service, is has negligible performance impact and can be gathered in production. Datamime's dataset generation and search can then be performed by either a third party (e.g., a research group) or the operator of the production service itself. This enables many use cases, such as producing open-source benchmarks for the research community, or quickly producing benchmarks that can be shared with providers (e.g., processor and system designers) to guide their designs without revealing proprietary data. Datamime is publicly available at `https://datamime.csail.mit.edu`.

In summary, we make the following contributions:

- We introduce *data-centric benchmark generation*, a general technique to produce representative benchmarks for production workloads by synthesizing a representative dataset.
- We present Datamime, an implementation of data-centric benchmark generation. Datamime is the first technique to generate *representative* synthetic datasets by matching the performance profiles of production workloads.
- We evaluate Datamime on several datacenter workloads, showing that it is effective at generating representative datasets. We show that Datamime can match the profile distributions of the target application's key metrics. We also show that Datamime produces the dataset in few iterations, and can generate datasets with a wide range of performance profiles.

II. Background

Since many datacenter applications are publicly accessible (or have a similar open-source counterpart), creating a representative benchmark can be accomplished by running the application with a representative dataset. Unfortunately, companies are often apprehensive about releasing the data used in production environments due to confidentiality issues.

Given this limited access to production data, prior work has designed datacenter benchmarks in two different ways. On the one hand, numerous synthetic benchmark suites have been proposed [12, 16, 31, 54]. These suites combine a set of common datacenter applications along with publicly available datasets. On the other hand, prior work has proposed black-box workload cloning techniques [4, 5, 17, 27], which synthesize proxy benchmarks that mimic the behavior of production workloads. We now discuss the limitations of these two approaches.

A. Existing Benchmarks use Unrepresentative Datasets

With the rise of numerous cloud services over the past few decades, a number of cloud benchmark suites have been published. Cloudsuite [12] focuses on introducing a set of scale-out workloads that exhibit significantly different microarchitectural characteristics from traditional server workloads, such as limited instruction- and memory-level parallelism and large working set sizes. Tailbench [31] aggregates a set of representative latency-critical applications, providing

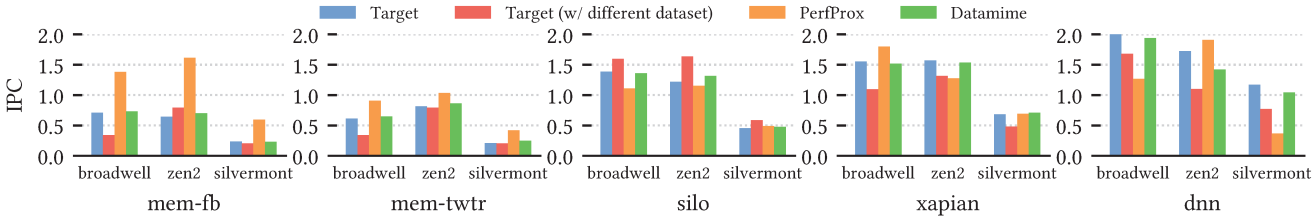

Figure 3: Accuracy comparison of five different target workloads against three other schemes. Each of the five plots shows the comparison of IPCs of different schemes against each target workload (`mem-fb`, `mem-twtr`, `silo`, `xapian`, `dnn`). For each group of bars in a given plot, we report the IPC measured on a system with different processor microarchitectures (Intel Broadwell, AMD Zen 2, and Intel Silvermont). The leftmost bar (blue) is the target workload's IPC, and the red bar is the IPC of the same application with a different dataset. We also report the IPCs of benchmarks generated by PerfProx and Datamime (orange and green) which mimic the microarchitectural characteristics of the target workload. See Sec. IV for details on our experimental setup, such as the target and different datasets used for the evaluation.

a testbench where it is possible to run these workloads with different network configurations. The DeathStarBench Suite [16] instead shifts its focus onto microservices: tens to hundreds of loosely-coupled collaborative services that form a single end-to-end application.

A major shortcoming of cloud benchmark suites is that the benchmarks are driven with input data that are not representative of datasets encountered in production environments. In part, this is because the benchmark suites use datasets that were never intended to be representative in the first place. For example, Cloudsuite and Tailbench both use YCSB [7], which is intended for performance benchmarking rather than being a representative workload.

In addition, even when benchmark designers drive the applications with traces or queries from real-world applications with anonymized customer data, the fact remains that these benchmarks cannot represent the wide range of different datasets that arise in production environments. For instance, analysis of key-value stores running on Facebook [51] and Twitter [57] have shown that production datasets differ widely in many dimensions, like their read/write ratio, hit rate, and key/value size distributions.

This discrepancy in the input dataset induces significant differences between the benchmark and the workload it tries to represent, making it difficult to use these benchmarks instead of the actual workloads for microarchitectural design-space exploration. For instance, consider Fig. 3, which extends the IPC comparison study in Fig. 1 to five target workloads. `mem-fb` and `mem-twtr` both use `memcached` as its application; the former uses a public dataset that closely matches a Facebook production service [3], and the latter uses a publicly available anonymized trace from Twitter's Twemcache [57]. `silo` is an in-memory database driven with a synthetic bidding dataset, `xapian` is a search engine driven with the default Tailbench dataset [31], and `dnn` is an object recognition neural network using the ResNet-50 model [20]. Each target workload (blue) and the *same application* running a different dataset (red) are evaluated on three systems with different microarchitectures (Intel Broadwell, AMD Zen 2, and Intel Silvermont).

Notice that for the `mem-fb` workload, running the same application with a different input can result in as much as 59% difference in the average IPC on the Broadwell processor, indicating that a representative input is crucial to producing accurate benchmarks. Datamime (green) is able to bridge this gap, generating a representative benchmark that results in only 4.1% difference in IPC averaged across all three systems. This trend clearly holds for the four other target workloads as well (see Sec. IV for details on our experimental setup).

B. Black-Box Workload Cloning Cannot Generate Representative Benchmarks

Prior work has proposed black-box workload cloning [4, 5, 17, 27] to automatically generate *proxy* benchmarks that mimic real-world applications. Black-box workload cloning operates on the assumption that neither the code nor the data of the target application can be shared publicly. Thus, workload cloning first profiles the target application to gather several key metrics that it seeks to replicate with the synthetic proxy. These often include instruction mix, basic block size, and memory access patterns. Based on this information, it generates a synthetic program that mimics these metrics.

A common shortcoming of black-box workload cloning techniques is their inability to capture application-level behavior, instead opting to create a sequence of instructions that mimic the specific metrics of interest. For instance, Joshi et al. [27] create a graph of basic blocks where the transition probability is equal to how often the branch terminating each basic block was taken, disregarding the application context within which the branch was taken. This approach extends to other metrics, such as generating a sequence of instructions to match instruction mixes, creating data dependences, and generating synthetic memory access streams.

We observe that this limitation makes black-box cloning unable to produce benchmarks that match the original workload well, limiting its usefulness in microarchitectural design exploration. Fig. 3 shows the average IPC measurements of benchmarks created by PerfProx [41] (orange), a state-of-the-art black-box workload cloning technique, that mimic the behavior of the five corresponding target workloads.

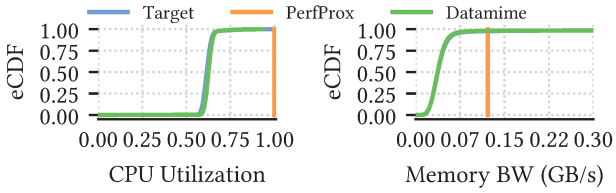

Figure 4: Comparison of the empirical cumulative distribution functions (eCDFs) of CPU Utilization and memory bandwidth usage of `mem-fb`. Each line shows the samples measured for each scheme against each metric: the target workload is `memcached` running a production dataset, and the two benchmarks are generated by PerfProx and Datamime, respectively.

The benchmarks are generated on the Broadwell machine, and validated on the AMD Zen 2 and Intel Silvermont processors. Notice that PerfProx is unable to generate an accurate proxy that matches the target workload's IPC across all microarchitectures, especially for certain workloads, such as `mem-fb`, where its IPC is up to 2.5× higher than the target workload's.

In addition, datacenter applications often have important time-varying behavior that workload cloning techniques fail to capture. Benchmarks should capture these transients as they heavily influence the important end-to-end metrics, such as the tail latency distribution that is shaped by intermittent long-latency requests [9, 29]. However, black-box cloning only captures *average* statistics, such as average IPC and predominant memory access stride, and reduce the original application down to a small binary that only mimics aggregate behavior, and is unable to mimic the different phases of activity within the real workload.

Fig. 4 shows that datacenter workloads have temporal changes in their behavior that black-box cloning fails to address. Each plot shows the empirical cumulative distribution functions (eCDF) plotted against the CPU utilization (left) and memory bandwidth usage (right). Each line shows the eCDF of a single scheme against the respective metrics: The Target (blue), which is `memcached` with a dataset that is representative of Facebook's production environment, and the two benchmarks generated by Datamime (green) and PerfProx (orange) for the target workload. We observe that the target workload exhibits meaningful distributions in both metrics, which PerfProx is incapable of imitating with its benchmark. In contrast, Datamime generates a benchmark that produces similar distributions for both metrics.

III. Datamime Design

We now describe the Datamime data-centric benchmark generator. Datamime creates representative benchmarks by automatically synthesizing datasets. Fig. 5 shows an overview of Datamime's design and usage flow.

The first step is profiling the target production workload to gather several key metrics. We choose a set of metrics that characterizes multiple facets of a program's behavior, such as instruction and data footprint, data locality, and request arrival rate. These include cache misses per kilo-instruction (MPKI), memory traffic, TLB misses, CPU utilization, and branch MPKI (Sec. III-A). In addition, we profile the sensitivity of the workload to cache capacity by measuring the last-level cache (LLC) MPKI and overall IPC with respect to various cache partition sizes.

Table I. Metrics captured by the Datamime Profiler.

Category	Profiled Metrics
Instruction Footprint	Instruction Cache MPKI Instruction TLB MPKI
Data Footprint	L1 Data Cache MPKI L2 Cache MPKI Data TLB MPKI
Cache Sensitivity	Last-level Cache MPKI Curve (across cache sizes) IPC Curve (across cache sizes)
Miscellaneous	Branch MPKI CPU Utilization Memory Bandwidth Usage (in GB/s)

Datamime uses two other inputs to produce a representative benchmark: a program (which should be the same or similar to the target workload's program), and a *dataset generator* for that program. The dataset generator takes a set of parameters and should be able to generate datasets that produce a wide range of microarchitectural behaviors. Although the dataset generator needs to be built for each program, we describe a systematic procedure for constructing useful generators (Sec. III-B).

Datamime uses the profiles from the target workload to search a space of possible dataset configurations (Sec. III-C). Datamime formulates this search as an *optimization* problem. Each iteration of the optimizer runs the program with a dataset generated from specific parameters, profiles it, and evaluates the error between the resulting profile and that of the target workload. The optimizer iterates over the dataset parameters, exploring the space to find a set of parameters that minimizes the error with the target workload.

A. Profiling

Datamime begins by gathering detailed profiles of the target workload. To make sure that we create a benchmark that is truly representative, the chosen metrics must be diverse enough to capture the various application behaviors, such as instruction-level parallelism, memory access patterns, and frequency of branch mispredictions.

To capture the overall application behavior, we measure 10 metrics of interest, listed in Table I. We choose a wide variety of metrics such that we capture multiple aspects of the target workload's behavior and to avoid overfitting to the microarchitecture of the machine used to generate the benchmarks. We track the instruction footprint of the workload by measuring the instruction cache misses and instruction

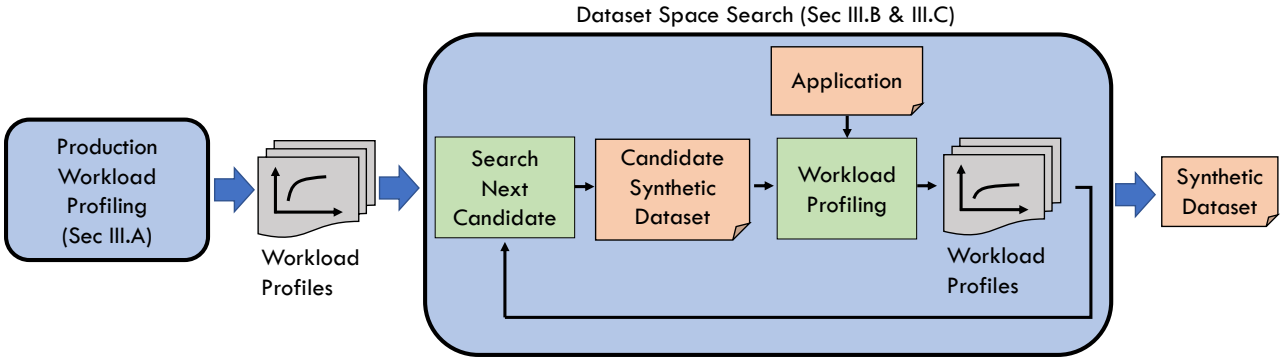

Figure 5: Structure of Datamime, with corresponding sections of the paper labeled.

TLB misses. Data footprint is captured by measuring the miss rate at all cache levels and also measuring data TLB misses. Other key behaviors are also tracked, including branch mispredictions (which relate to data-dependent code paths), CPU utilization (which relate to request arrival rates and service time distributions in cloud workloads), and memory bandwidth.

We use hardware performance counters to measure these metrics over a sufficient period of time. Importantly, we profile entire *distributions* of samples (e.g., producing a histogram of IPC over time), not just average values. Distributions capture not only the average behavior over the application's lifetime, but also the variability in the application's activity across phases of execution. Capturing this time-varying behavior is important for datacenter workloads because infrequent events, such as a sudden burst of requests that induce server-side queueing, can dominate the overall service-level performance [9].

In addition to the above metrics, we capture the memory access pattern of the application by measuring its sensitivity to cache capacity. To measure these we use Dynaway [11], a technique that measures LLC miss curves and IPC curves at low overhead. These curves capture the sensitivity of the workload to different cache sizes.

Our profiling technique generates enough samples to construct accurate distributions of these metrics. Each profiling iteration takes 2–4 minutes, depending on the CPU utilization of the profiled workload (lower CPU utilization requires additional time to capture stable profiles). This profiling time is important because Datamime not only profiles the target workload, but also the program and dataset for each iteration of the search. Datamime converges in under 200 iterations, so the whole search process takes a few hours.

B. Parameterizing the Dataset

Requirements for dataset parameterization: Having a good dataset generator is a key requirement for Datamime to produce accurate benchmarks. Ideally, the dataset generator should produce datasets that exhibit a wide range of performance behaviors and distributions to capture the full range of behaviors of the production workload. With an appropriate dataset generator, Datamime's problem is reduced to finding a set of parameters that produce similar behavior to the target workload. However, if the dataset generator is insufficiently broad, it may miss key characteristics of the target workload's dataset, and Datamime will not be able to find a close-enough dataset no matter how much it searches.

Importantly, Datamime uses an efficient optimizer that handles high-dimensional search spaces well (Sec. III-C). Thus, dataset generators can use a large number of parameters to capture a wide range of behaviors, and when writing a generator, it is not necessary to skimp on the number of parameters to keep search cost reasonable.

In addition, parameter selection does *not* require any knowledge of the target workload's dataset. Writing a generator simply requires some basic understanding of the target workload's program so that varying dataset parameters results in a wide range of behaviors. For example, when choosing the set of parameters for a web search engine, we use the fact that processing a request consists of retrieving the set of documents associated to the search term. Thus, the distributions of document sizes and search terms are the natural dataset parameters.

Systematically choosing parameters: Though building a dataset generator may seem like an ad-hoc process at first, by building several generators we have realized that the process is nearly identical across applications, and can be systematized, at least for request-driven applications. Our approach consists of parameterizing both requests and program data.

The first set of parameters that we include are ones that characterize the rate and types of requests for the application. This can be as simple as just adding the request rate (in queries per second, QPS) as a parameter for some applications with uniform requests, such as a search engine. For programs that have heterogeneous request types, we add parameters that control the ratio of these requests, such as the GET/SET request ratio for `memcached`, or the ratio of database transaction types for `silo`.

Next, our parameter selection strategy for the data itself largely depends on the *structure* of the resulting dataset. We categorize a dataset as being *unstructured* if there are no restrictions in the organization of the data, such as the

keys and values of a key-value store. Conversely, structured datasets have specific schemas that we must adhere to, such as the organization of tables in a relational database or the structure of layers in a neural network.

For unstructured data, we opt the simple approach of creating the datasets following certain *distributions* about their sizes. For instance, consider `memcached`, which simply consists of two different datatypes: keys and values. We start with the assumption that their sizes are normally distributed, then add the mean and standard deviation of each as the parameters to be adjusted.

For structured data, we take an application-specific strategy, since each application has different requirements in terms of how its dataset should be formed. For some, this is as simple as scaling up an existing synthetic dataset with the required structure or taking a subset of a publicly existing dataset. For example, in the case of `silo`, an in-memory database, where the dataset structure is tightly linked to the request types, we choose the scaling of an existing synthetic dataset (TPC-C). In the case of `xapian`, which searches through text documents that have certain properties like word frequency and sentence structure, we select a subset of documents from a public web crawl using the document length as its parameter. Finally, some applications such as `dnn`, a DNN-as-a-service application, have datasets which can be composed of simple building blocks, each of which can be a parameter of the dataset (in the case of `dnn`, this would be the number of layers for each layer type).

Refining parameters iteratively: Beyond the above process, the user can observe how well the produced dataset matches the behavior of the target workload, and add, change, or remove parameters if Datamime does not converge to a sufficiently accurate benchmark. In our experience, following the above parameter selection process is sufficient for most workloads (`memcached`, `silo`, `dnn`), and no refinement was required. For `xapian`, we had to refine some parameters (specifically, generalizing the distributions of document sizes and search terms) as our initial set of parameters did not imitate the target workload's behavior.

We observe that parameterization and subsequent generation of an application's synthetic dataset requires a modest amount of time from the benchmark designer. The only significant manual work required is the parameterization step, as exploring the search space is carried out automatically by Datamime (Sec. III-C). In our experience, all of our workloads took less than a week of manual work to determine a suitable set of parameters that resulted in a well-matching dataset.

C. Searching the Parameterized Dataset Space

Error Model: To search the optimal set of parameters, we must first define the goodness of a given dataset. We do this by defining the error in performance profiles between the synthesized benchmark and the target workload. Note that Datamime aims not only to match the averages of the performance metrics of interest, but also to match the *distributions* of the performance profiles between the production workloads and the corresponding benchmarks. A matching distribution indicates that the benchmark mimics both the long-term average behavior of the production workload and the short-term variations in its performance.

We use the Earth Mover's Distance (EMD) [46] metric to quantify the error between two distributions. Given two distributions with the N samples, we first define the cost of moving a single sample a unit distance from its original value as $\frac{1}{N}$. Then, the EMD between the two distributions is defined as the minimal total cost of moving samples from one distribution such that it matches the other. In the case of one-dimensional samples, this is simply the area between the two cumulative distribution functions [21]. Although other choices for measuring the error in distributions may be viable [8, 39], we found EMD to work well in our setting.

Given a set of parameters $\mathbf{p} = \{p_1, p_2, ..., p_n\}$ within the space of possible parameters P, we define the overall error $E_{\hat{\mathbf{p}}}(\mathbf{p})$ between the synthetic and target profiles $\mathbf{p}$ and $\hat{\mathbf{p}}$ by summing the pairwise EMDs between individual profiles p_i and $\hat{p}_i$:

$$E_{\hat{\mathbf{p}}}(\mathbf{p}) = \sum_i EMD(p_i, \hat{p}_i) \tag{1}$$

We normalize each metric to lie within $[0,1]$, and weight all metrics equally to make sure one mismatched metric does not dominate this error.

Formulating the optimization problem: It is prohibitively expensive to do a direct search for the right parameters, e.g., using random or grid-based search. First, the search space is extremely large due to the number of parameters we wish to add for dataset generation. For example, even if we allow only integer values, `memcached` has 329 trillion possible combinations of parameter values. In addition, evaluating each set of parameter values takes a non-trivial amount of time—typically a couple of minutes to obtain a sufficient number of samples—so evaluating even a few thousand points in the search space would take days to complete.

To search this large space efficiently, we formulate the search as an optimization problem, where the goal is to find a set of parameters $\mathbf{p}$ that minimize the overall error $E_{\hat{\mathbf{p}}}$:

$$\mathbf{p} = \underset{\mathbf{x}}{\operatorname{argmin}} E_{\hat{\mathbf{p}}}(\mathbf{x}) \tag{2}$$

The major challenges with solving this optimization problem are that the objective function is *black-box, expensive, and noisy*. The objective function being a black-box means that the function's analytical form is unknown. Thus, a gradient can only be approximated by measuring points in the solution space. Expensiveness indicates that each function evaluation takes a long time, so we would like to find a suitable solution with few evaluations. Finally, because the microarchitectural characteristics that we are measuring are subject to variations even with the same dataset, two function evaluations at the same point may result in different errors.

The black-box nature of the problem excludes simpler gradient-based optimization techniques such as gradient descent [33] because the convexity of the solution space is not guaranteed and exact gradient information is unavailable. Each function evaluation is expensive since profiling takes 2–4 minutes to complete. This rules out global optimization algorithms such as Simulated Annealing [32] and Genetic Algorithms [23] since such global optimization techniques typically require a large number of function evaluations [26].

The challenges we outlined naturally guided us towards using Bayesian Optimization, often used for problems with a noisy, expensive black-box objective function [26, 50]. Bayesian optimization has been successfully used in settings where the requirements are similar to those we face, such as hyperparameter tuning for machine learning models [13, 49, 50], robotics [2, 35], and finding optimal job co-location strategies in datacenters [45]. In addition, Bayesian optimization has been shown to handle optimization problems with up to 20 dimensions [15], which significantly eases the task of selecting dataset parameters as adding a few ineffectual ones will not significantly degrade the performance of the optimizer. We find that, in practice, the Bayesian Optimizer is very effective at generating a suitable dataset in a few hundred function evaluations (see Sec. V-D).

After the optimizer provides the next set of dataset parameters to evaluate, we generate the dataset and, together with the application, run the entire benchmark. We generate the same set of profiles as the target workload for the benchmark, and measure the EMD error. The measured error between the two set of profiles is then fed back to the optimizer, which selects the next point to evaluate in the search space.

D. Limitations of Dataset Generation

Our dataset generators do not produce values that match those of the target workload (e.g., some use randomly generated strings, others use a corpus of open-source data). This will introduce inaccuracies on systems that use value-dependent techniques, such as cache or memory compression. (Luckily, few systems use these features.) Solving this problem in general is hard, because capturing and mimicking the values of the target workload would leak proprietary data. However, dataset generators could be extended in technique-specific ways that allow them to remain representative without revealing program data. For instance, to evaluate the impact of cache compression techniques, Datamime could profile the compression ratio of the target workload's memory snapshots, and the dataset generator could then produce similarly compressible data. We leave this to future work.

IV. Methodology

Evaluation Platforms: Our evaluation uses three systems with different processor microarchitectures, listed in Table II We generate all of our benchmarks with PerfProx and Datamime on the 8-core Intel Broadwell system. For cross-microarchitecture IPC validation (Fig. 3), we use two different machines: a 32-core AMD Zen2 machine and a 8-core Silvermont machine. We choose these machines as they are quite different from the Broadwell machine: the Zen2 machine is more recent, has deeper buffers, and uses different predictors; and Silvermont is a low-power core with limited pipeline width and small OOO buffers.

TABLE II. SPECIFICATIONS OF THE SYSTEMS USED IN THE EVALUATION.

Cores	8 Xeon D-1540 cores (Broadwell), 2.0 GHz
L1 caches	32 KB per core, 8-way set-associative, split D/I
L2 cache	256 KB, core-private, 8-way set-associative
L3 cache	12 MB, shared, 12-way set-associative, inclusive, DRRIP policy [25, 55]; Way-partitioning with Intel CAT [22], supports 12 partitions
Memory	32 GB (2 × 16 GB DIMMs), DDR4 2133 MT/s
OS	Ubuntu 18.04, Linux kernel version 4.15
Cores	32 Ryzen ThreadRipper PRO 3975WX cores (Zen2), 3.50 GHz
L1 caches	32 KB per core, 8-way set-associative, split D/I
L2 cache	512 KB, core-private, 8-way set-associative
L3 cache	128 MB, 16 MB per chiplet, 16-way set-associative
Memory	256 GB, DDR4 3200 MT/s
OS	Ubuntu 20.04, Linux kernel version 5.4
Cores	8 Atom C2750 cores (Silvermont), 2.40 GHz
L1 caches	24 KB/32 KB per core, 8-way set-associative, split D/I
L2 cache	1 MB, core-private, 8-way set-associative
Memory	32 GB, DDR3 1600 MT/s
OS	Ubuntu 18.04, Linux kernel version 4.15

Experimental Setup: We use hardware performance counters to derive the metrics of interest as discussed in Sec. III-A, and use Intel CAT [22] to derive IPC and LLC MPKI curves. All performance counters are measured at 20 M cycle intervals, and we measure the IPC and memory traffic curves every 10 B cycles to minimize its effect on application performance. We disable TurboBoost to prevent performance fluctuations [30], and make sure that no other processes or threads are co-located on the same core as the profiled thread.

We generate the PerfProx benchmarks with the original code, which the PerfProx authors graciously provided. When generating the benchmarks, we follow the exact steps and configurations laid out in the original paper. Note that the PerfProx paper [41] reports generally lower errors in its IPC and other metrics compared to our findings. We attribute this to the fact that PerfProx was originally evaluated on a different set of database applications, whereas we target workloads that have a wider range of behavioral differences.

TABLE III. SUMMARY OF DATASET PARAMETERS FOR EACH WORKLOAD.

Workload	Parameters
memcached	QPS, get/set ratio, key size mean and standard deviation, value size mean and standard deviation
silo	QPS, # warehouses, ratio of TPC-C transactions (new order, payment, delivery, order status, stock level)
xapian	QPS, Zipfian skew, term frequency, average document length
dnn	QPS, # 3×3 conv. layers, # 3×3 strided conv. layers, # maxpool layers, # FC layers, # output channels of first layer

For all the target workloads and benchmarks generated by Datamime, the client and the server both reside on the same machine, and communicate either through the network stack (`mem-fb`, `mem-twtr`) or through shared memory using the Tailbench integrated configuration [31] (`silo`, `xapian`, `dnn`). We run Datamime's optimizer for 200 iterations, and choose the set of parameters determined as the lowest-cost point by the optimizer to generate the final synthetic dataset. Datamime runs each iteration sequentially on a single machine in our setup. Parallelizing the search process is possible by using parallel Bayesian optimization [6, 48, 56], but the serial process is fast enough, so we leave this to future work.

Applications: We evaluate Datamime using four applications: `memcached` (in-memory caching), `silo` (in-memory database), `xapian` (search engine), and `dnn` (object recognition). For each application, we choose an existing public dataset as the target workload (two for `memcached`) that we aim to match with the dataset generated by Datamime. Table III summarizes our choice of parameters for the dataset generators. We describe each application, its target dataset(s), and our selection of synthetic dataset parameters below:

`memcached` [14] is an open-source distributed in-memory key-value store widely used in industry. `memcached` is often deployed across hundreds of nodes to service millions of queries per second, where each node caches a portion of frequently accessed data. `memcached`-based services are commonly used in production settings [3, 37, 57].

We target `memcached` running two different datasets: a dataset representative of Facebook's environment [3] (`mem-fb`), and an anonymized trace from Twitter's Twemcache [57] (`mem-twtr`). We use mutilate [34] to generate requests for `memcached` according to the input dataset.

Parameters for the synthetic dataset include the QPS and get/set ratio for request distribution, and knobs to control the key and value size distributions, which we assume to be Gaussian. We use the same set of parameters to generate the synthetic datasets for both target workloads.

`silo` [53] is a fast transactional in-memory database. In-memory databases like `silo` are the backbone of online transaction processing workloads (OLTP) that have high throughput and low latency requirements [58].

The target workload for `silo` uses a synthetic bidding benchmark, where each transaction generates a bid on a random item in a table and, if larger than the current bid, overwrites the current entry. We parameterize the dataset by scaling the number of warehouses in a TPC-C benchmark, and varying the transaction ratios.

`xapian` [1] is an open-source search engine library allowing easy integration of indexing and search capabilities, and is used in popular websites (e.g., Debian web search) and integrated in multiple search applications (e.g., Recoll). Web search engines in production typically handle petabytes of data spread across thousands of leaf nodes [12, 31], and each node is responsible for handling a portion of the queries. We model our workload as a single leaf node in our setup.

The target workload for `xapian` uses the default input from Tailbench, an index of the 2013 English Wikipedia dump with a Zipfian query distribution. The synthetic dataset is constructed by indexing pages of a StackOverflow dump [52] whose sizes are within 50 bytes of the desired average document length. Queries are generated from a parameterized portion of all possible terms based on an upper limit of the term frequency, and we also control the Zipfian skew of the query distribution.

`dnn` is an object recognition workload using convolutional neural networks. Neural network inference in the cloud using CPUs has been a popular model for cloud providers due to their flexibility and availability [42]. We implement a simple inference setup using the PyTorch C++ frontend [43], and use the Tailbench harness to set up the client-server interface. We drive the application using validation images from the ImageNet library [47].

The dataset of interest in this workload is the *neural network model*, not the images themselves. The target workload for `dnn` uses a pre-trained ResNet-50 [20] model as its dataset, a popular network for object recognition. We construct a synthetic dataset using four frequently-encountered layer types: 3×3 convolution, 2×2 maxpool, 3×3 strided convolution, and fully-connected layers. We vary the number and position of each layer, except for the locations of the fully-connected layers, which are always positioned at the end of the network. In addition, we vary the number of output channels of the first layer to vary the total number of features at each layer.

In addition to the datasets mentioned above, Fig. 1 and Fig. 3 report results with alternative, publicly available datasets for each application (shown in red bars). For `memcached` and `silo`, these are Tailbench's defaults: YCSB-A and TPC-C. For `xapian`, we use an index constructed from a portion of the StackOverflow dump [52]. And for `dnn`, we use a pre-trained ShuffleNet model [36].

V. EVALUATION

We now analyze the effectiveness of Datamime in creating representative benchmarks. We evaluate Datamime's ability

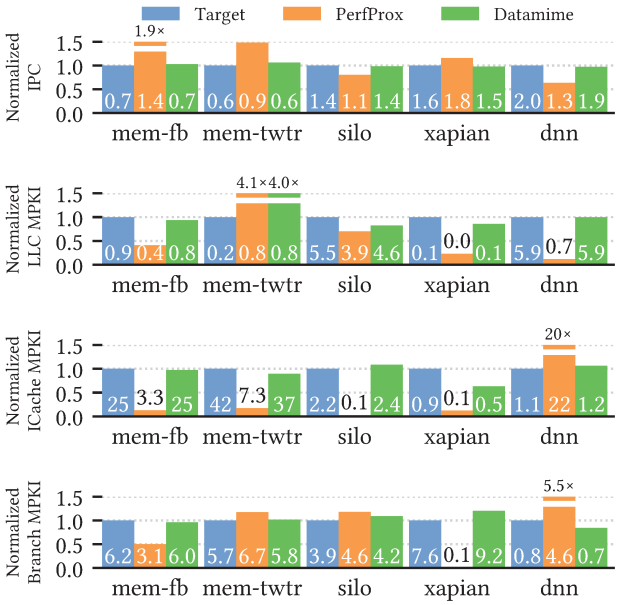

Figure 6: Comparison of performance metrics of PerfProx and Datamime, normalized to the Target metrics for each workload. The absolute value for each metric is labeled on each bar.

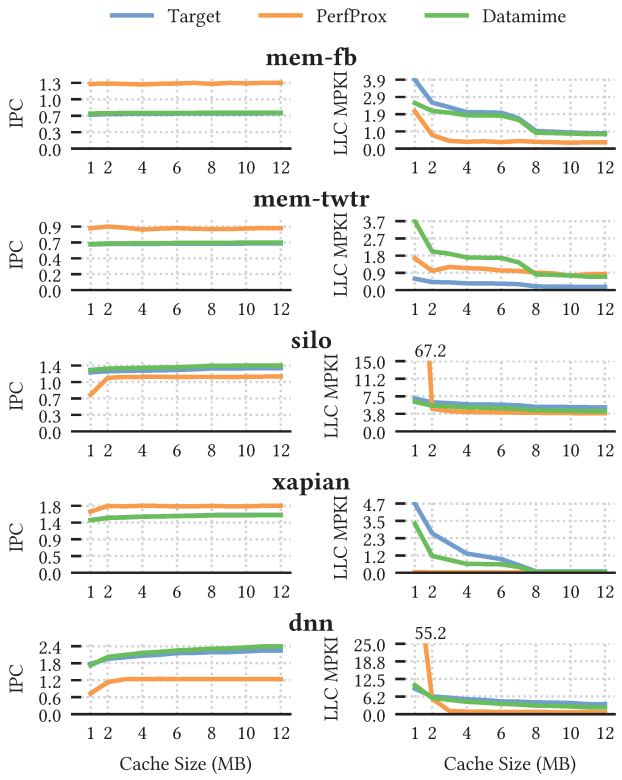

Figure 7: Comparison of IPC and LLC MPKI curves between the Target workload, PerfProx benchmark, and benchmark generated by Datamime. Each workload/benchmark is allocated a cache size between 1MB to 12MB in 1MB increments, and we measure the IPC and LLC MPKI for each allocation.

to match various performance profiles on five different target workloads, and compare it against PerfProx. We also evaluate Datamime's ability to accurately match performance profile distributions. Finally, we analyze various performance aspects of Datamime: its speed of convergence and the possible range of profiles it can generate.

A. Datamime matches performance profiles much better than black-box cloning

Fig. 6 summarizes the effectiveness of Datamime's benchmark generation strategy compared to PerfProx. Each graph reports results for a different microarchitectural metric: instructions per cycle (IPC), last-level cache misses per kilo-instruction (LLC MPKI), instruction cache MPKI (ICache MPKI), and branch MPKI. Within each graph, each group of bars reports results for a single application. Each group consists of three bars: Target is the workload we wish to mimic, PerfProx is the black-box cloning technique we compare against, and Datamime is our workload generation technique. The height of each bar is the average of the performance metric normalized to Target. Thus, values close to 1.0 are better. In addition, the bottom of each bar lists the absolute (non-normalized) average value for each metric.

Datamime matches the target workloads much more closely than PerfProx. Averaged across workloads, Datamime results in 3.2% *mean absolute percentage error* on IPC, which is defined as $|IPC_{target} - IPC_{benchmark}|/IPC_{target}$ averaged across workloads. By contrast, PerfProx has an 42.9% mean absolute percentage error on IPC. Datamime also matches other metrics well, for which we measure the *mean absolute error* defined as $|Metric_{target} - Metric_{benchmark}|$ averaged across workloads. Datamime achieves an error of 0.34 for LLC MPKI, 1.16 for ICache MPKI, and 0.47 for branch MPKI. This is a much smaller gap compared to the error of 1.62 for LLC MPKI, 16.3 for ICache MPKI, and 3.22 for branch MPKI that PerfProx achieves.

Datamime is particularly effective at matching the *key metrics* that most influence the target workload's behavior. These include the high instruction cache misses for `mem-fb` and `mem-twtr`, the high LLC MPKI for `silo`, and the branch MPKI for all workloads. In contrast, PerfProx is not able to match the code behavior effectively except for two workloads (`mem-twtr` and `silo`) for which it only matches the branching behavior well.

Looking at individual workloads, we see that Datamime matches `mem-fb` particularly well. This is expected, as its dataset is unstructured and therefore completely defined by our distribution parameters. It is noteworthy that we match `mem-fb` when our assumed value distribution (Gaussian) is different from that of the target workload's dataset (generalized Pareto) [3], indicating that exactly matching all the characteristics of the target dataset is not needed to match its performance profile.

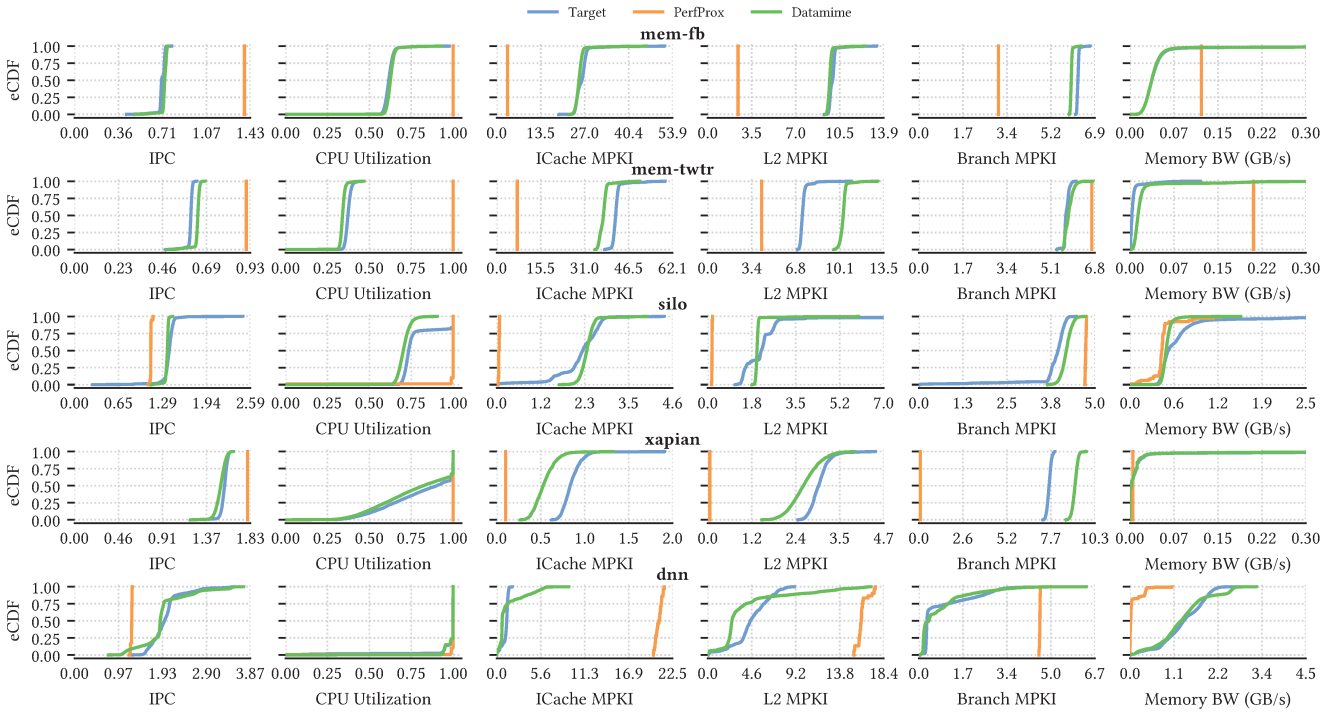

Figure 8: Detailed plots of the *distributions* of the performance counter samples we measure across all workloads. Each plot shows the empirical cumulative distribution function (eCDF) of the measured profile for the three schemes: the Target workload, benchmark generated by PerfProx, and Datamime's synthetic benchmark.

There are some metrics where Datamime has high relative errors, such as `mem-twtr` and `xapian`'s LLC MPKI and `xapian`'s ICache MPKI. However, note that in absolute terms these metrics are small in the target workload, and their impact on end-to-end performance is limited. Although we do not particularly bias the search to prioritize matching metrics where absolute values are large, by including IPC as one of the metrics, the search naturally gives more importance to metrics that have a large influence on end-to-end performance.

Fig. 7 shows the IPC and LLC MPKI of each workload when different cache sizes are allocated to the worker thread for the Target workload and the benchmarks generated by PerfProx and Datamime. Note that `dnn` has a higher IPC and lower LLC MPKI at 12MB compared to Fig. 6 because cache capacity is allocated to a single profiled worker thread.

In general, Datamime is able to match both the shapes and values of the curves. Even in cases where Datamime is unable to match the exact curve (e.g., `xapian`'s LLC MPKI curve), Datamime matches the *shape* of the curve, indicating that the generated benchmark is able to match the memory access pattern and cache sensitivity of the target workload. In contrast, PerfProx is unable to match the IPC and LLC MPKI curves for most applications. In particular, it produces benchmarks with sharp cache cliffs at 1MB for `silo` and `dnn`, which the target workloads do not exhibit.

B. Datamime matches performance profile distributions

Fig. 8 shows the *statistical distributions* of several key metrics: IPC, CPU utilization, ICache MPKI, L2 Cache MPKI, branch MPKI, and memory bandwidth. Each plot shows the empirical cumulative distribution function (eCDF, i.e., cumulative histogram) of the samples (taken at 20 MCycle intervals) from the Target workload, and PerfProx's and Datamime's benchmarks. Note that a low slope of the eCDF curve indicates a high variance in the measured samples.

First, we see that even for metrics where PerfProx matches the target reasonably well, such as the IPC for `silo` and `xapian`, it cannot match their distributions. PerfProx does not produce meaningful distributions because its behavior is *static* over time. This is expected, as workload cloning techniques mimic average behavior. This trend is more readily apparent in certain metrics, such as branch MPKI and memory bandwidth in `dnn`, which exhibit wide variances.

In contrast, Datamime clearly follows the time-varying behavior of the target workloads, such as the variance in CPU utilization due to long requests. Similar to the memory traffic curves in Fig. 7, Datamime matches the shapes of the distributions even when it does not match the values, such as the branch MPKI for `xapian`.

Like for averages, Datamime has larger errors on metrics whose absolute values are small and have little impact on end-to-end performance.

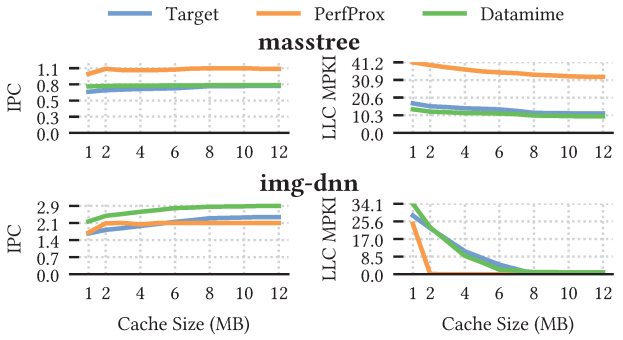

Figure 9: Comparison of IPC and LLC MPKI curves between Target, PerfProx and Datamime for `masstree` and `img-dnn` when Datamime uses *different*, but functionally similar applications in its search (`memcached` for `masstree`, and `dnn` for `img-dnn`).

C. Case study: Targeting a workload with a different program

So far we have assumed that the target workload's application is publicly available, and produced benchmarks using the same program as the target workload. We now show that Datamime is able to match a workload's behavior in certain dimensions *even when using a different program*.

We target two workloads from Tailbench: `masstree`, a key-value store driven with YCSB, and `img-dnn`, a handwriting recognition engine that uses a deep neural network-based autoencoder driven with the MNIST dataset [10]. For Datamime, we choose the programs that most closely match the behavior of the target workloads: `memcached` for `masstree`, and `dnn` for `img-dnn`. We use the same dataset generators as before.

Fig. 9 shows the IPC and LLC MPKI curves of the Target workload and the two benchmarks generated by PerfProx and Datamime. Since the source applications of Datamime roughly imitates the high-level behavior of the target workloads, Datamime is able to match the IPC and LLC MPKI curves of `masstree` *even though the code is different*. Datamime also matches the LLC MPKI curve of `img-dnn`, but overshoots the IPC due to an inherent trade-off in how well Datamime can match these two metrics using `dnn`. To verify this, we reran Datamime's search giving higher weight to IPC, which resulted in an accurate IPC curve match for `img-dnn` at the expense of a worse LLC MPKI match.

There are other metrics, such as the ICache and branch misses, which Datamime cannot match as accurately due to the differences in the code. For instance, `masstree` has lower cache misses in general because it is designed to be cache-optimized [38], whereas `memcached` is not. These differences are summarized in Table IV, along with comparisons against the proxies for each workload generated with PerfProx. Even with the large differences in the detailed metrics, Datamime outperforms PerfProx in matching the important end-to-end metrics such as IPC (mean absolute percentage error of 8.6% versus 19.4%) and LLC MPKI (mean absolute error of 1.73 versus 10.9).

TABLE IV. MEASURED METRICS OF THE TARGET WORKLOAD, PERFPROX, AND DATAMIME FOR `masstree` AND `img-dnn`. DATAMIME'S RESULTS ARE WITH BENCHMARKS GENERATED WITH A *different* APPLICATION THAN THE TARGET WORKLOAD (`memcached` AND `dnn`, RESPECTIVELY).

Metric	`masstree`			`img-dnn`		
	Target	PerfProx	Datamime w/ different program	Target	PerfProx	Datamime w/ different program
IPC	0.79	1.05	0.79	2.25	2.11	2.63
LLC MPKI	11.4	32.7	10.6	0.45	0.02	3.07
CPU Util.	0.57	1.00	0.97	0.39	1.00	1.00
Branch MPKI	14.9	27.7	4.96	0.53	0.09	0.20
ICache MPKI	1.20	0.24	16.3	0.53	0.11	0.21
L1D MPKI	8.80	14.1	34.2	26.1	0.13	9.17
L2 MPKI	6.14	14.7	15.7	1.70	0.05	1.96
ITLB MPKI	0.13	0.02	0.50	0.04	0.02	0.03
DTLB MPKI	2.24	0.52	19.5	0.61	0.87	0.22
Mem. Bw (GB/s)	0.62	4.29	0.97	0.05	0.01	0.96

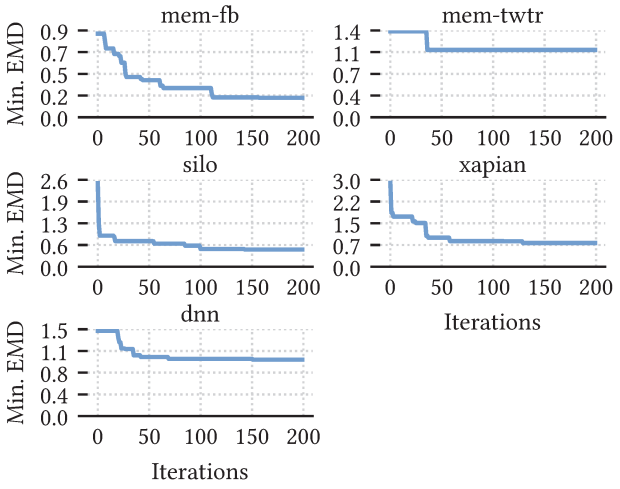

Figure 10: Total EMD error of all performance profiles as a function of the number of iterations performed by the optimizer.

D. Speed of convergence

Fig. 10 shows the rate at which the *minimum* observed EMD (Datamime's error metric) decreases with respect to the number of iterations. The EMD is the area between the CDFs of the target workload and the synthesized benchmark for each metric, where the x- and y-axes are *normalized* to lie between zero and one, by dividing them by maximum x and y values observed. For example, in Fig. 8, the ICache MPKI plot for `xapian` has an EMD value 0.23, as the area between the target and Datamime CDFs is 23% of the plot's area. Fig. 10 reports the sum of EMDs across all 10 evaluated metrics, so for example, a total EMD of 0.7 corresponds to an average EMD per metric of 0.07, i.e., 7% of the area. An iteration in this context refers to a single evaluation of a set of parameters, feeding the resulting EMD to the optimizer, and choosing the next set of parameters to evaluate.

When evaluated for 200 iterations, we observe that Datamime approaches to within 90%, 0%, 48.4%, 22.6%, and 4.7% of the achievable minimum EMD in 50 iterations for `mem-fb`, `mem-twtr`, `silo`, `xapian`, `dnn` respectively, indicating that our technique is effective at generating a fairly

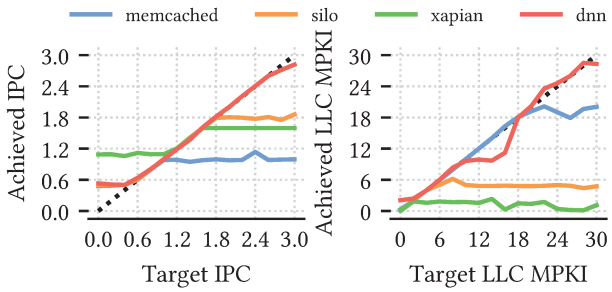

Figure 11: Sweep of achievable target IPC and LLC MPKI for Datamime with the four publicly available applications we use in our setup. The x-axis is the IPC/LLC MPKI we ask Datamime to produce with each application, and the y-axis shows the actual IPC/LLC MPKI Datamime achieves.

representative dataset in a short amount of time. Note that the absolute time taken per iteration largely depends on the number of LLC MPKI and IPC curve samples taken, as these have long profiling intervals (10 B cycles). In practice, we find that taking 11 samples is an effective tradeoff between accuracy and speed, resulting in 2–4 minutes per iteration. This translates to 6–13 hours for Datamime to run 200 iterations and produce each benchmark.

E. Range of possible performance profiles generated by Datamime

It is important for dataset generators to produce a wide range of performance profiles. A wide range lets Datamime match a wide variety of production workloads. We can measure this range by using Datamime to try to match an arbitrary value for one or more metrics, rather than the performance profiles of a particular workload.

Fig. 11 shows the results of such an experiment. Each graph shows results for a single target metric: IPC (left), and LLC MPKI (right). Within each graph, each line shows results for a separate application. For each line, we sweep the target metric over a wide range of values, shown in the x-axis, and the y-axis denotes the actual value that Datamime achieves. For each experiment we configure Datamime to *only* match the target metric so that we measure the maximum achievable range for each metric, sweeping the target metric at 15 evenly spaced points. As long as Datamime can match the target, each line falls on the $y = x$ line.

Fig. 11 shows that Datamime matches a varying range of IPCs across workloads: between 0.48 and 1.14 for `memcached`, 0.50 and 1.86 for `silo`, 1.05 and 1.60 for `xapian`, and 0.50 and 2.82 for `dnn`. The relatively small range of IPC for `memcached` and `xapian` can be attributed to the fact that their operations are much more uniform (cache misses and branch mispredictions limit IPC regardless of the input). In contrast, `dnn` can have widely different computations based on the structure of its input neural network, which results in its wide range of IPCs.

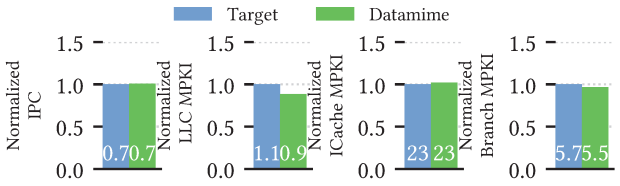

Figure 12: Key performance metrics of Datamime relative to Target for `mem-fb` with the server and client configured to communicate over the network.

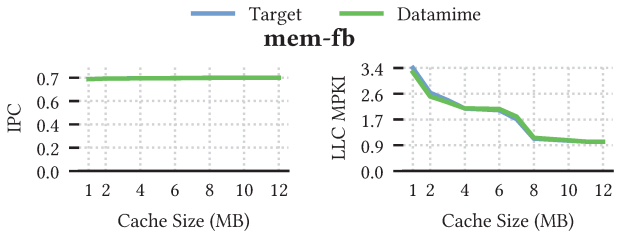

Figure 13: IPC and LLC MPKI curves of Target and Datamime for the networked configuration.

Datamime also matches a varying range of LLC MPKIs. LLC MPKIs ranges between 0.29 and 20.1 for `memcached`, 0.04 and 6.18 for `silo`, 0.03 and 2.32 for `xapian`, and 2.07 and 28.5 for `dnn`. Just like IPC, the range of LLC MPKI for each workload depends on the workload's underlying structure. For example, for `xapian`, the most popular queries dominate execution, so the most frequently accessed documents are cached on-chip, resulting in limited memory traffic regardless of the dataset.

F. Datamime on multi-machine benchmarks

So far, we have reported results for single-machine experiments, where the benchmark and request generator run under the same node. This setup is convenient, but does not capture the original workload's interactions with network I/O. We now show that Datamime remains accurate in a multi-machine setting. We focus on Memcached, as it is the benchmark with the shortest requests among our applications (tens to hundreds of microseconds per request) and therefore it is the most affected by the additional network latency and overheads. We modify our experimental setup for `mem-fb` where we host the server (Memcached) and our load generator (mutilate) on two separate machines.

Fig. 12 shows the averages of several key metrics for the target workload and the synthesized benchmark, similar to Fig. 6. Datamime synthesizes a benchmark that can closely mimic the complex networking interactions of the original workload, resulting in 1% mean absolute percentage error for IPC and 0.12 mean absolute error for LLC MPKI. Just like in the single-machine configuration, Fig. 13 also shows that Datamime closely matches the IPC and LLC MPKI curves of the target workload.

VI. Additional Related Work

We now discuss related work not covered in Sec. II.

A. Synthetic Cloud Benchmark Suites

Several synthetic cloud benchmark suites exist today. Cloudsuite [12] gathers a set of scale-out and throughput-oriented benchmarks, and analyzes the microarchitectural requirements of cloud applications. BigDataBench [54] focuses on evaluating a wide variety of data types and a broader set of workloads. Tailbench [31] includes a set of latency-critical applications, and introduces an evaluation methodology focused on tail latency. DeathStarBench [16] introduces a set of workloads that use the microservices model, consisting of tens to hundreds of loosely coupled tiny services instead of one or few large monolithic applications.

All of these benchmark suites are constructed by selecting a set of datacenter applications and using a set of publicly available datasets to drive them. Benchmark suites often use synthetic datasets, such as TPC-C or public web crawls, that are not representative of production data. Using anonymized production datasets or traces can temporarily elide this issue, but a single dataset rarely is enough to model the variety of data encountered in production settings [3, 51, 57], nor does it stay representative over time [3, 51]. Datamime resolves these issues by constructing workloads that accurately imitate the target workload's performance profiles. This makes it easy to keep benchmarks up-to-date with production workloads: it simply requires generating a new benchmark from a recent performance profile.

B. Black-Box Workload Cloning

The seminal work in black-box workload cloning is Bell and John [5], which produces small testcases from an application's performance statistics using statistical flow graphs. Although the original goal was to generate short, representative test cases that could be simulated and quickly compared to the target applications, it has spawned a line of work that leverages the fact that profile-based benchmark synthesis hides information about the target application's code.

Joshi et al. [27] improves upon prior work by using microarchitecture-independent models to capture program characteristics, allowing the synthetic workload to preserve the application characteristics across microarchitectures. Benchmaker [28] profiles workloads at a much coarser granularity, collecting average statistics over the entire program instead of at basic-block granularity. Ganesan et al. [17] incorporate memory-level parallelism in characterizing the target workload. WEST [4] generates benchmarks that focus on accurately mimicking data cache behavior.

While black-box workload cloning does have the benefit of producing short benchmarks that are quick to evaluate, they fail to capture the high-level program behavior and the temporal changes in program characteristics. Workload cloning techniques reduce the target workload to a few set of average statistics, a process that loses much of the crucial information about the program. In contrast, Datamime uses the same or similar application as that used in the target workload, thereby preserving the program structure and more successfully imitating its overall behavior.

C. Synthetic Data Generators

Given the inaccessibility of production datasets, prior work has also proposed *data generators* that produce structured and unstructured data using statistical modeling of the target dataset [18, 24, 40, 44]. These techniques generate synthetic data by first modeling existing datasets (either real-world or synthetic), and then generating datasets that follow this model, such as the distribution of topics in a text document [40].

These data generation techniques are different in purpose to the dataset generators used in Datamime, and do not seek to generate representative benchmarks. The major shortcoming of prior data generators is that they focus on matching the characteristics of the real *dataset*, rather than the characteristics of the *resulting workload*. Without guiding the data generation process with how the dataset would change the application performance, the representativeness of the dataset cannot be guaranteed. In addition, all prior data generators either do not care about the representativeness of their dataset [18, 24], or never validate their synthetic data against production workloads [40, 44]. This makes it difficult to rely on these datasets to provide accurate performance characteristics of production workloads. In contrast, Datamime uses profiling information to guide its dataset generation, and validates its resulting performance profiles against those of the production workload.

In addition, prior data generators need an accurate model of the target dataset in generating the synthetic data, which introduces the danger of leaking information about the production dataset. For instance, The text data generator of BigDataBench [54] uses detailed information about an existing text dataset such as the distribution of words and topics. If the production dataset is used as the input to the modeling phase, which is needed if the data generator wishes to produce representative datasets, the resulting synthetic dataset may leak information about the confidential data *because* it mimics its characteristics.

Lastly, Dataset generation using statistical techniques can be *complementary* to Datamime's profile-guided generation. Datamime can confine the possible set of synthetic datasets to those that match the target dataset's statistical properties, which would significantly speed up its search process. We note that this approach requires access to the target dataset (or its relevant statistical properties), and necessitates the dataset to be easily modeled statistically. For instance, it would be difficult to incorporate statistical modeling into creating a `silo` benchmark since key aspects such as the ratio of different transactions is difficult to model statistically.

VII. Conclusion

We have introduced data-centric benchmark generation, a new insight that shows that using publicly available code along with synthetically generated datasets is an effective approach to generating representative benchmarks. We have presented Datamime, a technique that leverages this insight. Datamime uses publicly available applications along with synthetically generated datasets to create benchmarks that closely mimic cloud workloads. We have shown that Datamime creates benchmarks that are much more representative of target workloads compared to prior black-box cloning techniques, matches the temporal behavior of workloads, and also matches key metrics such as IPC and LLC MPKI even when the target workload's program is unavailable. Datamime is publicly available to enable the community to easily produce representative benchmarks and ultimately accelerate architecture research and design.

VIII. Acknowledgments

We sincerely thank Quan Nguyen, Victor Ying, Yifan Yang, Axel Feldmann, Nikola Samardzic, Fares Elsabbagh, Shabnam Sheikhha, Robert Durfee, Nithya Attaluri, Kendall Garner, Joel Emer, and the anonymous reviewers for their helpful feedback. We thank Lizy John for graciously sharing the PerfProx code, and we thank Abhishek Dhanotia for his feedback on earlier versions of Datamime. This work was supported in part by a Facebook Research Award and by NSF grant SHF-1814969. Hyun Ryong Lee was partially supported by a Kwanjeong Educational Foundation fellowship.

References

[1] "Xapian project," https://github.com/xapian/xapian.

[2] R. Antonova, A. Rai, and C. G. Atkeson, "Deep kernels for optimizing locomotion controllers," in *Proceedings of the 1st Annual Conference on Robot Learning*, 2017.

[3] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *Proceedings of the ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, 2012.

[4] G. Balakrishnan and Y. Solihin, "WEST: cloning data cache behavior using stochastic traces," in *Proceedings of the 18th IEEE international symposium on High Performance Computer Architecture (HPCA-18)*, 2012.

[5] R. H. Bell Jr. and L. K. John, "Improved automatic testcase synthesis for performance model validation," in *Proceedings of the International Conference on Supercomputing (ICS'05)*, 2005.

[6] E. Contal, D. Buffoni, A. Robicquet, and N. Vayatis, "Parallel gaussian process optimization with upper confidence bound and pure exploration," in *Proceedings of Machine Learning and Principles and Practice of Knowledge Discovery in Databases, European Conference (ECML PKDD)*, 2013.

[7] 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 (SoCC)*, 2010.

[8] D. A. Darling, "The Kolmogorov-Smirnov, Cramer-von Mises tests," *The Annals of Mathematical Statistics*, vol. 28, no. 4, 1957.

[9] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, vol. 56, no. 2, 2013.

[10] L. Deng, "The MNIST database of handwritten digit images for machine learning research," *IEEE Signal Processing Magazine*, vol. 29, no. 6, 2012.

[11] N. El-Sayed, A. Mukkara, P. Tsai, H. Kasture, X. Ma, and D. Sanchez, "KPart: A hybrid cache partitioning-sharing technique for commodity multicores," in *Proceedings of the 24th IEEE international symposium on High Performance Computer Architecture (HPCA-24)*, 2018.

[12] M. Ferdman, A. Adileh, Y. O. Koçberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: a study of emerging scale-out workloads on modern hardware," in *Proceedings of the 17th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XVII)*, 2012.

[13] M. Feurer, A. Klein, K. Eggensperger, J. Springenberg, M. Blum, and F. Hutter, "Efficient and robust automated machine learning," in *Proceedings of the 28th International Conference on Neural Information Processing Systems (NIPS)*, 2015.

[14] B. Fitzpatrick, "Distributed caching with memcached," *Linux journal*, vol. 124, 2004.

[15] P. I. Frazier, "A tutorial on Bayesian optimization," *CoRR*, vol. abs/1807.02811, 2018.

[16] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, and C. Delimitrou, "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems," in *Proceedings of the 24th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXIV)*, 2019.

[17] K. Ganesan, J. Jo, and L. K. John, "Synthesizing memory-level parallelism aware miniature clones for SPEC CPU2006 and ImplantBench workloads," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2010.

[18] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger, "Quickly generating billion-record synthetic databases," in *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 1994.

[19] J. Hauswald, M. A. Laurenzano, Y. Zhang, C. Li, A. Rovinski, A. Khurana, R. G. Dreslinski, T. N. Mudge, V. Petrucci, L. Tang, and J. Mars, "Sirius: An open end-to-end voice and vision personal assistant and its implications for future warehouse scale computers," in *Proceedings of the 20th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XX)*, 2015.

[20] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[21] K. Henderson, B. Gallagher, and T. Eliassi-Rad, "EP-MEANS: an efficient nonparametric clustering of empirical probability distributions," in *Proceedings of the 30th Annual ACM Symposium on Applied Computing (SAC)*, 2015.

[22] A. Herdrich, E. Verplanke, P. Autee, R. Illikkal, C. Gianos, R. Singhal, and R. R. Iyer, "Cache QoS: From concept to reality in the Intel® Xeon® processor E5-2600 v3 product family," in *Proceedings of the 22nd IEEE international symposium on High Performance Computer Architecture (HPCA-22)*, 2016.

[23] J. H. Holland, "Genetic algorithms," *Scientific American*, vol. 267, no. 1, 1992.

[24] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The HiBench benchmark suite: Characterization of the MapReduce-based data analysis," in *Proceedings of the 26th International Conference on Data Engineering Workshops (ICDEW 2010)*, 2010.

[25] A. Jaleel, K. B. Theobald, S. C. Steely Jr., and J. S. Emer, "High performance cache replacement using re-reference interval prediction (RRIP)," in *Proceedings of the 37th annual International Symposium on Computer Architecture (ISCA-37)*, 2010.

[26] D. R. Jones, M. Schonlau, and W. J. Welch, "Efficient global optimization of expensive black-box functions," *Journal of Global Optimization*, vol. 13, no. 4, 1998.

[27] A. Joshi, L. Eeckhout, R. H. Bell Jr., and L. K. John, "Performance cloning: A technique for disseminating proprietary applications as benchmarks," in *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, 2006.

[28] A. Joshi, L. Eeckhout, and L. K. John, "The return of synthetic benchmarks," in *2008 SPEC Benchmark Workshop*, 2008.

[29] S. Kanev, J. P. Darago, K. M. Hazelwood, P. Ranganathan, T. Moseley, G. Wei, and D. M. Brooks, "Profiling a warehouse-scale computer," in *Proceedings of the 42nd annual International Symposium on Computer Architecture (ISCA-42)*, 2015.

[30] S. Kanev, K. M. Hazelwood, G. Wei, and D. M. Brooks, "Tradeoffs between power management and tail latency in warehouse-scale applications," in *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, 2014.

[31] H. Kasture and D. Sanchez, "Tailbench: a benchmark suite and evaluation methodology for latency-critical applications," in *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, 2016.

[32] S. Kirkpatrick, C. D. Gelatt Jr, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, no. 4598, 1983.

[33] Y. LeCun, D. Touresky, G. Hinton, and T. Sejnowski, "A theoretical framework for back-propagation," in *Proceedings of the 1988 Connectionist Models Summer School*, 1988.

[34] J. Leverich and C. Kozyrakis, "Reconciling high server utilization and sub-millisecond quality-of-service," in *Proceedings of the EuroSys Conference (EuroSys)*, 2014.

[35] D. J. Lizotte, T. Wang, M. H. Bowling, and D. Schuurmans, "Automatic gait optimization with Gaussian process regression," in *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI'07)*, 2007.

[36] N. Ma, X. Zhang, H. Zheng, and J. Sun, "ShuffleNet V2: practical guidelines for efficient CNN architecture design," *CoRR*, vol. abs/1807.11164, 2018.

[37] S. Madappa and S. Enugula, "Evolution of application data caching: From RAM to SSD," https://netflixtechblog.com/evolution-of-application-data-caching-from-ram-to-ssd-a33d6fa7a690, 2018 (accessed May 11, 2022).

[38] Y. Mao, E. Kohler, and R. T. Morris, "Cache craftiness for fast multicore key-value storage," in *Proceedings of the EuroSys Conference (EuroSys)*, 2012.

[39] F. J. Massey Jr, "The Kolmogorov-Smirnov test for goodness of fit," *Journal of the American Statistical Association*, vol. 46, no. 253, 1951.

[40] Z. Ming, L. Chunjie, W. Gao, R. Han, Q. Yang, L. Wang, and J. Zhan, "BDGS: A scalable big data generator suite in big data benchmarking," *CoRR*, vol. abs/1401.5465, 2014.

[41] R. Panda and L. K. John, "Proxy benchmarks for emerging big-data workloads," in *Proceedings of the 26th International Conference on Parallel Architectures and Compilation Techniques (PACT-26)*, 2017.

[42] J. Park, M. Naumov, P. Basu, S. Deng, A. Kalaiah, D. S. Khudia, J. Law, P. Malani, A. Malevich, N. Satish, J. M. Pino, M. Schatz, A. Sidorov, V. Sivakumar, A. Tulloch, X. Wang, Y. Wu, H. Yuen, U. Diril, D. Dzhulgakov, K. M. Hazelwood, B. Jia, Y. Jia, L. Qiao, V. Rao, N. Rotem, S. Yoo, and M. Smelyanskiy, "Deep learning inference in Facebook data centers: Characterization, performance optimizations and hardware implications," *CoRR*, vol. abs/1811.09886, 2018.

[43] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Z. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An imperative style, high-performance deep learning library," in *Proceedings of the 32nd International Conference on Neural Information Processing Systems (NeurIPS)*, 2019.

[44] T. Rabl, M. Frank, H. M. Sergieh, and H. Kosch, "A data generator for cloud-scale benchmarking," in *Proceedings of the Second TPC technology conference on Performance evaluation, measurement and characterization of complex systems (TPCTC'10)*, 2010.

[45] R. B. Roy, T. Patel, and D. Tiwari, "SATORI: Efficient and fair resource partitioning by sacrificing short-term benefits for long-term gains," in *Proceedings of the 48th annual International Symposium on Computer Architecture (ISCA-48)*, 2021.

[46] Y. Rubner, C. Tomasi, and L. J. Guibas, "A metric for distributions with applications to image databases," in *Proceedings of the Sixth International Conference on Computer Vision (ICCV-98)*, 1998.

[47] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. S. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet large scale visual recognition challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, 2015.

[48] A. Shah and Z. Ghahramani, "Parallel predictive entropy search for batch global optimization of expensive objective functions," in *Proceedings of the 28th International Conference on Neural Information Processing Systems (NIPS)*, 2015.

[49] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, and N. D. Freitas, "Taking the human out of the loop: A review of bayesian optimization," *Proceedings of the IEEE*, vol. 104, no. 1, 2015.

[50] J. Snoek, H. Larochelle, and R. P. Adams, "Practical bayesian optimization of machine learning algorithms," *Advances in neural information processing systems*, vol. 25, 2012.

[51] A. Sriraman and A. Dhanotia, "Accelerometer: Understanding acceleration opportunities for data center overheads at hyperscale," in *Proceedings of the 25th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXV)*, 2020.

[52] Stack Exchange Inc., "Stack exchange data dump," https://archive.org/details/stackexchange, 2018 (accessed May 11, 2022).

[53] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden, "Speedy transactions in multicore in-memory databases," in *Proceedings of the 24th Symposium on Operating System Principles (SOSP-24)*, 2013.

[54] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, C. Zheng, G. Lu, K. Zhan, X. Li, and B. Qiu, "BigDataBench: A big data benchmark suite from internet services," in *Proceedings of the 20th IEEE international symposium on High Performance Computer Architecture (HPCA-20)*, 2014.

[55] H. Wong, "Intel ivy bridge cache replacement policy," https://perma.cc/M59C-HPBN, 2013 (accessed May 11, 2022).

[56] J. Wu and P. I. Frazier, "The parallel knowledge gradient method for batch bayesian optimization," in *Proceedings of the 29th International Conference on Neural Information Processing Systems (NIPS)*, 2016.

[57] J. Yang, Y. Yue, and K. V. Rashmi, "A large scale analysis of hundreds of in-memory cache clusters at Twitter," in *Proceedings of the 14th USENIX symposium on Operating Systems Design and Implementation (OSDI-14)*, 2020.

[58] H. Zhang, D. G. Andersen, A. Pavlo, M. Kaminsky, L. Ma, and R. Shen, "Reducing the storage overhead of main-memory OLTP databases with hybrid indexes," in *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2016.