2DIO: Configurable and Cache-Accurate Trace Generation for Storage Benchmarking

Yirong Wang
Northeastern University
Boston, Massachusetts, USA
wang.yiron@northeastern.edu

Isaac Khor
Northeastern University
Boston, Massachusetts, USA
khor.i@northeastern.edu

Peter Desnoyers
Northeastern University
Boston, Massachusetts, USA
P.Desnoyers@northeastern.edu

Abstract

We introduce 2DIO, a microbenchmark creating cache-accurate, stressful I/O traces. While existing tools are limited to generating traces with well-behaved, *concave* hit ratio curves, 2DIO produces ones with tunable complex cache behaviors, particularly performance *cliffs* and *plateaus*.

Our framework encodes a workload as a compact parameter triplet, capturing both short-term recency and long-term frequency. This parsimonious parameterization allows researchers to easily translate individual adjustments into predictable cache effects across various eviction policies, and enables the parameter space to be "swept" for exhaustive exploration of desired cache behavior, or to mimic real traces by calibrating parameters to match observed behaviors.

The tuned parameters are portable, meaning if the scale of the system under evaluation changes, so too will the footprint and length of the trace, while the relative cache behaviors are preserved.

Evaluations demonstrate 2DIO's ability to generate traces across a continuum of "what-if" cache behaviors and to reproduce real-world ones with high accuracy.

CCS Concepts: • **General and reference → Performance**; • **Information systems → Hierarchical storage management**.

Keywords: Caching, Trace generation, Performance evaluation, Benchmarking

ACM Reference Format:
Yirong Wang, Isaac Khor, and Peter Desnoyers. 2026. 2DIO: Configurable and Cache-Accurate Trace Generation for Storage Benchmarking. In *European Conference on Computer Systems (EUROSYS '26), April 27–30, 2026, Edinburgh, Scotland Uk.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3767295.3769391

1 Introduction

Storage system research requires both measuring the performance of storage systems, and comparing these measured results against other systems. These systems are often large and complex, used by complex applications. Performance is affected not only by system hardware and configuration, but by application or workload characteristics, so measurements are inherently tied to the workload used when making them. Storage systems exhibit complex workload-dependent behaviors, and thus the choice of workload is key to measurements which can be used to compare and predict performance of these systems.

In an ideal world, these measurements would be: (1) predictive, allowing us to accurately estimate the performance of a system (or the relative performance of different systems) in future executions; (2) generalizable, providing predictions (especially comparative ones) which hold over as wide a range of applications and configurations as possible; (3) reproducible, i.e. different researchers should be able to make the same measurement and compare results, and (4) easy to perform, at an effort and cost which is not excessive relative to the system under test.

1.1 Benchmark Types

We begin by reviewing three ways to evaluate storage: (a) real workloads, i.e. the target application itself; (b) trace replay, recorded operations performed under a real workload and reproducing them with more or less fidelity to the original, and (c) synthetic workloads, with algorithmically-generated behaviors intended to mimic either real workloads or some aspect of their behaviors.

Real applications. In certain cases, e.g. HPC procurement, real application benchmarks are the "gold standard", giving the exact answer needed; however they pose a number of deficiencies for storage research. In particular results are not generalizable; if user A tests a system on workload A, while user B tests a different one on workload B, little is learned about system A vs B.

Trace replay. Trace replay offers a way to test different systems with identical, real-world workloads. Individual I/O operations are recorded on live systems, with trace corpuses like CloudPhysics[35] and AliCloud [3] offering diverse workloads. Standard (e.g. fio [4]) or custom replay tools are used to replay these traces against a system under test, allowing the original workload to be repeated on demand. But modern trace corpuses are large and cumbersome (70 GB and 700 GB for CloudPhysics and AliCloud), making

EUROSYS '26, Edinburgh, Scotland Uk

ACM ISBN 979-8-4007-2212-7/26/04
https://doi.org/10.1145/3767295.3769391

exhaustive replay impractical, and new corpuses are rare due to high procedural barriers to releasing potentially sensitive data, leading to long delays before new applications (e.g. LLMs) are reflected in available traces.

Synthetic traces. Synthetic trace generators that mimic real workloads address many issues with trace replay. Without the need for huge trace libraries, synthetic generators can be readily integrated into many tools.

Conceptually the task of creating such a generator is straightforward: measure the workload, encode it in a probabilistic model, and generate new traces from that model. With parameters capturing the key workload features, one can systematically explore the entire space rather than selecting arbitrary points from a set of opaque real traces.

But what characteristics to measure and reproduce? The answer seems simple: those that affect the system under test. If performance depends on I/O size, record and reproduce its distribution; if it differs between reads and writes, capture that ratio; if it varies with seek distance, model that as well.

If the system being evaluated is a cache, one would argue that the hit ratio curve (HRC), i.e., the hit ratio as a function of cache size, is the most critical. Although generators such as fio support frequency-skew models (Zipf, Pareto, Zoned) that influence the HRC, we show that this alone is far from sufficient to reproduce the HRCs observed in real workloads.

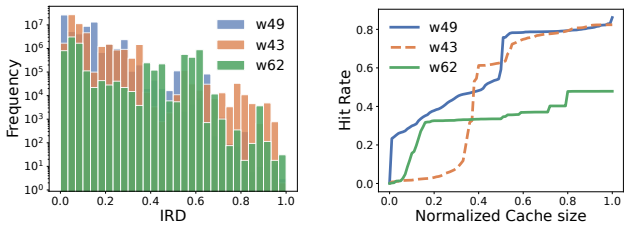
1.2 Cache-accurate Synthetic Traces

Workload generators are typically not used to predict the behavior of systems, but rather to compare them. An example is the evaluation sections of papers published in this conference, where data storage papers frequently use fio, Filebench [10], and trace replay to compare their system to prior work. These evaluations use (a) reproducible workloads, allowing experiments to be compared, (b) multiple workloads covering a range of possible application behaviors, and (c) at least some realistic workloads, mimicking real application behaviors. If system A out-performs B on most or all the tests in a properly-done evaluation, the community tends to believe this indicates a high probability that A will out-perform B in the field.

For systems where performance is heavily influenced by cache hit rate, the cacheability of this workload is important, as otherwise our benchmarks may measure miss performance when hits would be seen in the field, or vice versa.[1]

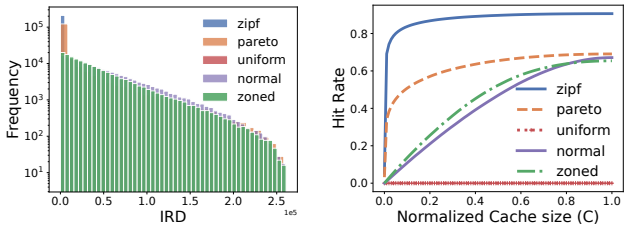
Frequency distributions. The state of the art in reproducing the cacheability of workload appears to be item *frequency* models, using either a parameterized distribution (typically Zipf) or empirically-measured ones from existing workloads [16, 24, 29, 31, 36, 37, 42]. The origin of this approach is Denning's *Independent Reference Model* (IRM) [1]: each arrival is assumed to be an independently weighted choice from the set of possible items.

[1]This was an issue in the IRCache web cache "bake-offs" several decades ago [15], where the use of uniform randomly-distributed workloads disadvantaged some vendors who had used more advanced caching algorithms.



(a) IRD histogram (b) LRU hit ratio

Figure 1. Several CloudPhysics traces showing diverse hit rate behavior. Cache size is normalized to the trace footprint, i.e. the total number of unique blocks accessed in the trace.



(a) IRD histogram (b) LRU hit ratio

Figure 2. IRD distributions and hit rate behavior for fio-generated synthetic traces.

IRM can be accurate for e.g., content delivery networks (CDNs) where many independent sources are aggregated into a single stream, yet it misses the mark wildly for block storage systems, where typical workloads are comprised of a few highly correlated streams. For example, Fig. 1 demonstrates the behavior of three real-world block traces from the CloudPhysics corpus [34, 35]. Their inter-reference distance (IRD) histograms (Fig. 1(a)) feature prominent *spikes* and *holes*. When fed into a Least-Recently-Used (LRU) cache, these traces produce complex *non-concave* HRCs (Fig. 1(b)), including performance *cliffs*, where small cache increases lead to large gains, and *plateaus*, where additional cache provides little benefit. In contrast, Fig. 2 shows fio-generated workloads, whose IRD distributions are strictly decreasing, yielding *concave* HRCs, reflecting diminishing returns from added cache.

Recency distributions. *It is mathematically impossible to manipulate frequency distributions in a way that would produce* non-concave *LRU HRCs [33].* Whether an access will hit in an LRU cache is solely determined by the IRD since the preceding access to the same item, and, in particular, whether it was evicted during this interval. Since IRM has an equal probability of accessing a particular item at each reference, these IRDs will always be exponentially distributed, while non-concavity in HRC arises precisely because IRDs (e.g. under scan-like workloads) are not memoryless.

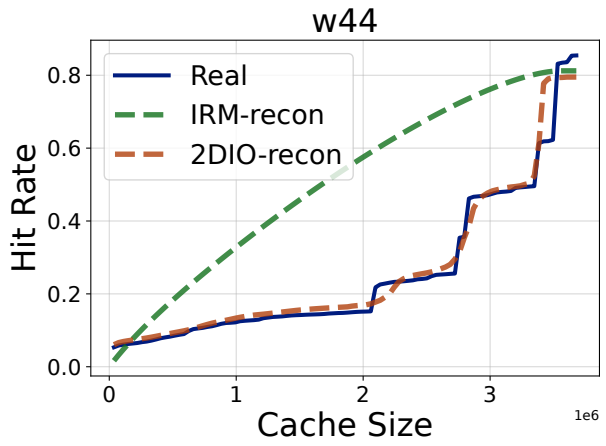


Figure 3. LRU HRCs for CloudPhysics w44: original trace (blue), 2DIO-generated trace (orange), and trace reconstructed using empirically-measured item frequency distribution (green). Cache size is measured in number of blocks.

The cache modeling community [9, 11–13, 21, 28, 34, 35, 39, 41] has long been using *recency* models such as stack distance (SD) [20] and IRD [7, 8] to characterize non-memoryless traffic. Despite their effectiveness in predicting the HRC of a given trace, we are not aware of any prior work using them in the reverse direction, i.e., generating trace(s) matching a target HRC.

To fill this gap, we introduce Gen-from-IRD, an algorithm that generates accesses based on a given IRD distribution. Traces generated via Gen-from-IRD precisely exhibit a target LRU HRC, and are far more accurate than IRM-generated traces for cache algorithms that use recency.

Frequency + recency distributions. We present 2DIO, a synthetic trace generator that faithfully produces (or reproduces) access frequency and recency.

Building on Gen-from-IRD, 2DIO encodes these characteristics into a compact parameter triplet, the *trace profile*, which creates or "counterfeit" workloads exhibiting desired performance *cliffs*/*plateaus*. Researchers can thus customize traces with precise target HRCs or mimic a real workload at various scales for diverse benchmarking tasks.

2DIO does this by merging recency (IRD) models for short-term behavior with frequency (IRM) models for long-term behavior. Its effectiveness is shown in Fig. 3, which displays the LRU HRC of CloudPhysics w44, as well as IRM- and 2DIO-reconstructions based on empirically measured characteristics. Although the IRM-reconstruction faithfully reproduces the access frequencies for different blocks in the trace, the result is a simple, concave HRC, with none of the performance anomalies seen with the original trace being reproduced. In contrast, the 2DIO-generated trace yields results very similar to the original, reproducing the performance *cliffs* and *plateaus* with only minor deviations.

This approach applies to broader areas, e.g., CDNs and web caches, where performance is measured by object or request hit rate regardless of object size.

Succinct parameterization is essential for tractability. For this purpose, 2DIO (a) approximates arbitrary non-memoryless IRD distribution as a coarse stepwise probability density function (PDF), and (b) limit the numeric distribution to relatively short IRDs, using a general IRM frequency distribution (e.g., Zipf) to approximate the tail of long-duration ones. This parsimonious parameterization creates standardized, reproducible trace profiles for consistent cross-system evaluation while allowing proprietary measurements of workload behavior to be distilled into a form which may be easier to make public, or shared among organizations.

1.3 Contributions

The contributions of 2DIO include:

1. encoding complex workload characteristics using a succinct parameter triplet, requiring only a handful of numerical values,
2. allowing the parameter space to be "swept" in experimentation, creating a continuum of complex LRU cache behaviors from exact reproduction of real-world patterns to hypothetical "what-if" scenarios,
3. reproducing these behaviors while arbitrary scaling both footprint and length, allowing authentic and stressful system benchmarking at various scales.

The rest of the paper is organized as follows: Sec. 2 covers background in workload models and real-world trace behaviors. Sec. 3 describes high level approach, introducing the two main algorithms. Sec. 4 explains design and implementation details. Sec. 5 evaluates 2DIO's fidelity, configurability, scalability, and discusses limitations. Sec. 6 reviews related research, and Sec. 7 concludes the paper.

2 Background

2.1 Workload models

Cache algorithms have long been characterized by how they make use of *recency* and *frequency* of cached items, i.e. the time since an item's last occurrence and the rate at which the item has occurred in the past, respectively. LRU uses only recency to make eviction decisions; least-frequently-used (LFU), on the other hand, uses only frequency. A broader spectrum of cache algorithms (e.g, First-In-First-Out (FIFO), CLOCK) are primarily recency-based, but respond to frequency as well.

In describing these models, we assume a footprint of M distinct items (e.g., a block of 4096 bytes); a cache that can hold at most C of them, never more; and a reference stream $r_1, r_2, \cdots$ of accesses to items in $\{0, 1, \cdots, M-1\}$. Following prior work [34, 35], we simulate the reference stream as a discrete-time process, measuring time t in distance (units of accesses). E.g. the distance between references r_4 and r_1 is 3.

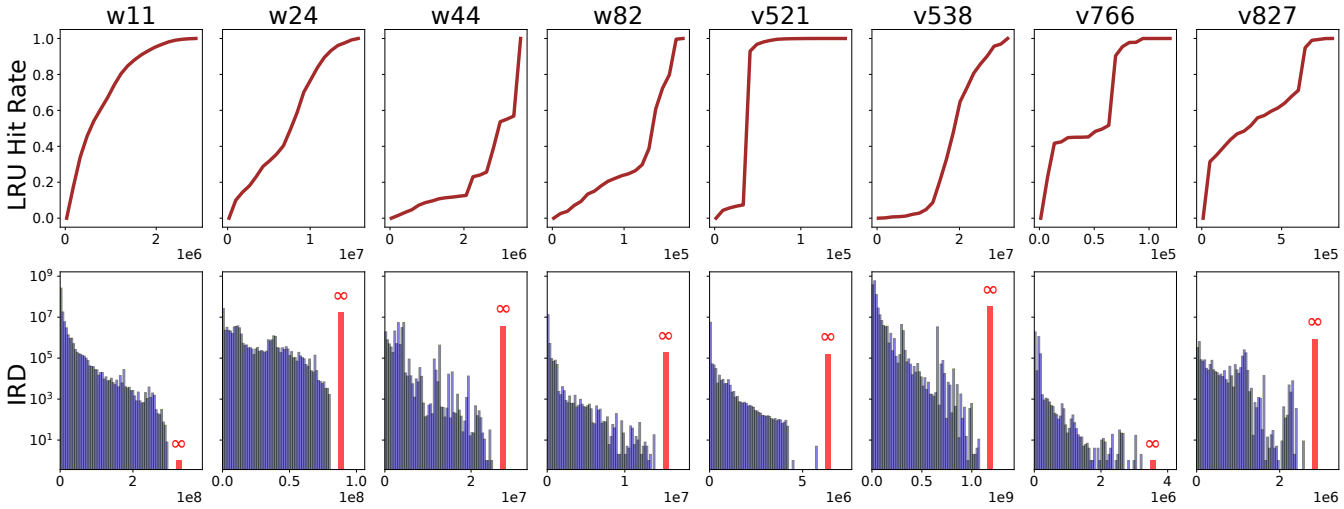


Figure 4. LRU HRCs and IRD histograms of real traces from AliCloud and CloudPhysics.

Independent reference model. IRM [1, 5] characterizes an access stream only by its item frequency; each item i is assigned with weights w_i, $i \in \{0, 1, \cdots, M-1\}$, and $\sum_i w_i = 1$. Hence access to i is chosen independently with probability w_i.

IRM workloads are widely used in microbenchmarks [4, 6, 10, 18], typically in its simplified forms. In the hot/cold model [25], $r_H M$ hot items are accessed at rate λ_H, and the remaining $(1 - r_H)M$ cold items are accessed at rate $1 - \lambda_H$; in the Zipfian model, λ_i corresponds to a Zipf distribution.

Dependent reference models. Alternately one can ignore frequency[2] and characterize an access stream by the distribution of *distances* between accesses to an individual item.

This approach has been widely used for *cache modeling*, i.e. predicting cache performance from workload characteristics [9, 13, 20, 21, 28, 40, 41].

Two measures have been used for this distance: *stack distance* [20] (SD) and *inter-reference distance* [7, 8] (IRD). Given two successive references r_i and r_j to the same item, the IRD refers to the distance between them in the reference stream (i.e. $j - i$), while the SD is the number of unique items referenced by the accesses separating them (i.e. $|\{r_{i+1}, r_{i+2}, \cdots, r_{j-1}\}|$).

AET Approximation. There is a direct correspondence between SD and LRU hit rate: if the SD between two references is C, then the second access will hit in any LRU cache of size C or larger.

This correspondence can be approximated to a correlation between IRD and LRU miss rate [11, 14]: if one knows the average time items stay in cache before eviction (AET), and the cumulative distribution function $F(t)$ of the request IRDs, one can approximate the cache size C (i.e., SD). It follows that

$$C = \int_0^{AET(C)} P(t)dt,$$

where $P(t) = 1 - F(t)$ is the probability that an item is not reused before t, and $AET(C)$ denotes the average eviction time of items in this cache.

The miss rate at cache size C is the probability that a reuse time is greater than $AET(C)$:

$$P_{miss}(C) = P(AET(C)).$$

This is known as Che's approximation [11], or AET approximation [14].

2.2 Real Block Trace Characteristics

IRM may well characterizes workloads seen by e.g. CDNs, where large numbers of independent sources are aggregated into a single request stream, "drowning out" correlations between references in a single stream. Block storage workloads are different. Their references typically driven by one or a few applications, preserving inter-reference correlations caused by application behavior.

Table 1 lists several AliCloud (v521, v538, v766, v827) and CloudPhysics (w11, w24, w44, w82) traces. This subset was chosen for subsequent analysis (Secs. 5.1, 5.3) because they exhibit diverse cache behaviors. Length and footprint are measured after converting each trace to PARDA [21] format[3].

Fig. 4 depicts their IRD histograms alongside the corresponding LRU HRCs. HRCs are seen to be highly *non-concave*, with steep sections (*cliffs*) and flat regions (*plateaus*). These correspond to the *spikes* and *holes* in the underlying IRD distribution. Holes starting at zero may be due to the OS

[2]Frequency does not affect LRU performance, but affects FIFO and CLOCK, and may have greater effect on more modern replacement algorithms.

[3]A sequence of 64-bit references without additional metadata, used for cache simulation.

Table 1. Trace subset chosen for subsequent analysis.

Trace ID	Length (N)	Footprint (M)
w11	296893045	2992519
w24	81762918	16487648
w44	25257814	3679382
w82	14198758	189785
v521	5974956	158018
v538	1204044775	33006370
v766	3335779	124146
v827	3198158	851527

buffer cache [38], which absorbs low-IRD accesses, while *spikes* may be caused by scan-like behavior.

A few of them display "well-behaved" IRM-like behaviors, e.g., w11 has a decreasing IRD histogram which corresponds to a concave HRC.

All of them show a noticeable percentage of "one-hit wonders" which are referenced once and never re-accessed over the duration of the trace[4]. In IRD measurements these accesses are recorded with an IRD of ∞.

3 2DIO Framework

The trace generation algorithm at the core of 2DIO can be considered a simple discrete-event simulation: each item repeatedly (a) generates one access to itself, (b) selects a sleep time t from the IRD distribution, and (c) sleeps for time t. Since we focus on distances between accesses rather than actual time, the algorithms always generate exactly one access for every position in the trace produced.

3.1 2DIO Generation from IRD

Algorithm 1 presents an effective way of generating a trace from a given IRD distribution.

Input. The algorithm takes the following input parameters:

(1) f – an IRD distribution
(2) M – footprint
(3) N – trace length

Output: Synthetic trace π_s.

Initialization: Draw M sleep times t from f. For each non-infinite t, assign a unique address a to it and add the pair to the priority queue (heap).

Trace generation. To generate a single reference, draw a t from the IRD distribution. If $t = \infty$ we draw an item from the singleton pool (addresses beyond M), otherwise we take the pair $\langle t_0, a_0 \rangle$ from the bottom of the priority queue, generate an access a_0 to the trace, and update a_0's sleep time to $t_0 + t$ in the priority queue. The algorithm continues until the trace is of length N.

[4]CloudPhysics and AliCloud traces are 7 and 30 days long; any access beyond this time period is unlikely to hit in cache.

Algorithm 1: Gen-from-IRD: generate a trace from a given inter-reference distance (IRD) distribution f.

Input: f, M, N
Output: π_s

```
Heap ← ∅
a ← 0
while Heap.size < M do
    t ~(i.i.d.) f
    if t ≠ ∞ then
        Heap.insert(< t, a >)
        a ← a + 1
π_s ← ∅
for j = 0 ... N − 1 do
    t ~(i.i.d.) f
    if t = ∞ then
        π_s.append(a)
        a ← a + 1
    else
        < t_0, a_0 > ← Heap.pop()
        π_s.append(a_0)
        Heap.replace(< t_0 + t, a_0 >)
return π_s
```

If one is only interested in shaping the LRU or other purely recency-based HRCs, then this algorithm is sufficient, for IRDs alone dictate the performance of these cache policies [8, 11]. Policies such as FIFO and CLOCK, however, also respond to frequency rather than recency alone. We thus describe an extension to Gen-from-IRD which induces tunable popularity skew.

3.2 2DIO Generation from IRD + IRM

Gen-from-IRD relies heavily on the accuracy of the input IRD distribution, often requiring it to be long enough to manifest a long tail. IRDs beyond the eviction time do not affect the *non-concave* HRC features and thus can instead be approximated by a succinct IRM input (e.g., Zipf (1.2)), while explicitly enforcing the frequency distribution.

We introduce Algorithm 2: Gen-from-2D. It is essentially derived from Gen-from-IRD, but merged with an IRM arrival process. In this process, item frequency follows distribution g, and each generation of such items is triggered with probability P_{IRM}. The process is visualized in Fig. 5. Merging a short-interval IRD distribution f with a long-interval IRM tail g lets the generator tune both dimensions, shaping the behavior of diverse cache policies.

Input: The algorithm takes 5 inputs:

(1) P_{IRM} – fraction of independent references
(2) g – an item frequency distribution
(3) f – IRDs represented as a piece-wise quantization
(4) M – footprint
(5) N – trace length

Algorithm 2: Gen-from-2D: generate a trace from the IRD distribution f with probability $1 - P_{IRM}$, and from the item-frequency distribution g with probability P_{IRM}.

Input: P_{IRM}, g, f, M, N
Output: π_s

```
Heap ← ∅
a ← 0
while Heap.size < M do
    t ~i.i.d.~ f
    if t ≠ ∞ then
        Heap.insert(< t, a >)
        a ← a + 1
π_s ← ∅
for j = 0 ... N − 1 do
    Random ~i.i.d.~ Uniform(0, 1)
    if Random < P_IRM then
        addr ~i.i.d.~ g
        π_s.append(addr)
    else
        t ~i.i.d.~ f
        if t = ∞ then
            π_s.append(a)
            a ← a + 1
        else
            < t_0, a_0 > ← Heap.pop()
            π_s.append(a_0)
            Heap.replace(< t_0 + t, a_0 >)
return π_s
```

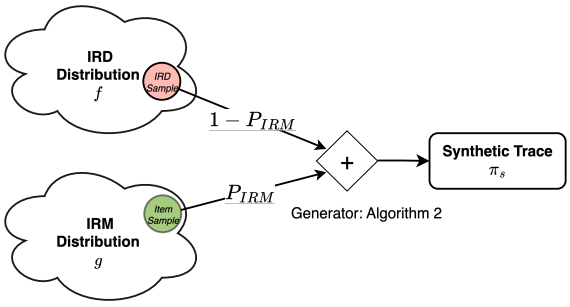


Figure 5. IRD + IRM trace generation uses Algorithm 2: references are drawn with probability P_{IRM} from an IRM process, and $1 - P_{IRM}$ from an IRD renewal process.

Output: Synthetic trace π_s.
Initialization: see Gen-from-IRD.
Trace generation: The algorithm is similar to Gen-from-IRD. However, at each iteration, with probability P_{IRM}, it chooses an item from g.

3.3 Parameters Customization

Following Algorithm 2, 2DIO takes in 5 inputs: P_{IRM}, g, f, M, and N. The first three we define as the *trace profile*, represented as a triplet $\theta = \langle P_{IRM}, g, f \rangle$. We refer to items generated by f as *dependent arrivals*, and those by g as *independent arrivals*. M and N are scale parameters which do not affect the (normalized) cache behavior.

The goal of 2DIO is to use a compact θ to create synthetic traces from scratch; it can also reproduce real traces by calibrating θ to match similar behavior. For both cases, 2DIO aims to approximate the target cache behavior, rather than reproducing it exactly, and that a given accuracy target may be met by more than one choice of θ.

Essentially, there is a trade-off between accuracy and succinctness: reproducing a real trace may require a long or empirically derived f (as in Fig. 3), whereas generating new behavior requires only a succinct f (e.g., fewer than 10 values with *spikes* at target percentiles), offering high tractability.

3.3.1 Configuring the IRD Distribution.

f specifies an IRD distribution from which the algorithm draws the sleep time for each item. A typical scenario is setting up a trace that has a cache benefit when the cache size is around e.g., 5% of the dataset. As proven by van den Berg and Towsley (1993) [33], this cannot be achieved by adjusting the frequency distribution; it requires shaping the underlying IRD distribution, e.g., inducing *spikes/holes* at specific values.

We note that this can be implemented with the AET approximation formulas [11, 14] presented in Sec. 2.1:

$$C = SD(\tau_c) = \int_0^{\tau_c} P(t)dt, \tag{1}$$

Since Eq. (1) is bijective, each mean eviction time τ_c on the IRD domain uniquely maps to a cache size C and vice versa.

The corresponding hit rate is given by:

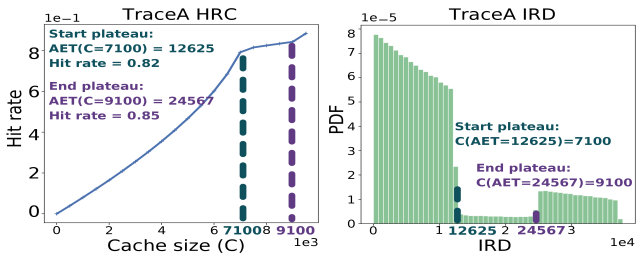
$$P_{hit}(C) = 1 - P(\tau_c), \tag{2}$$

whereby each τ_c also yields a unique hit rate for the stack distance C it approximates.

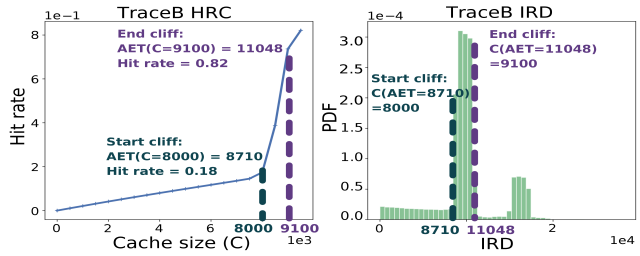
Fig. 6 visualizes such correspondence between a *hole* in the TraceA IRDs and a *plateau* in its HRC, as well as a *spike* in TraceB IRDs and a *cliff* in its HRC. To utilize this correspondence, 2DIO provides a model-accurate interface **fgen**. Let the IRD distribution $f(X = i)$ be a PMF with finite support $i \in \{1, 2, \cdots, k\}$. Let $\mathbf{fgen}(k, \mathcal{I}, \epsilon)$ be a function which generates f, assigning higher probability (*spikes*) to elements in the set $\mathcal{I}$ and lower probability (*holes*) to elements in its complement set $\overline{\mathcal{I}}$. The total probability mass of the *holes* sums to ϵ, resulting in the following distribution:

$$f(i) = \begin{cases} \frac{1-\epsilon}{|\mathcal{I}|}, & i \in \mathcal{I} \\ \frac{\epsilon}{k-|\mathcal{I}|}, & i \in \overline{\mathcal{I}} \end{cases} \tag{3}$$

2DIO then fit f over an auto-tuned IRD sample space (see Sec. 4.1) $\mathcal{S} := \{1, 2, \cdots, T_{max}\}$, result in a distribution with



(a) *plateau* in HRC at C_i to C_j ↔ *hole* in f at $AET(C_i)$ to $AET(C_j)$



(b) *cliff* in HRC at C_i to C_j ↔ *spike* in f at $AET(C_i)$ to $AET(C_j)$

Figure 6. correspondences between HRC *plateaus* & *cliffs* and IRD *holes* & *spikes*. Example synthetic traces Trace A and Trace B: LRU HRC (left), and corresponding IRD distributions (right).

spikes at IRD intervals:

$$\{[i \times \frac{T_{max}}{k},\ (i+1) \times \frac{T_{max}}{k}], \quad \forall i \in \mathcal{I}\}$$

and *holes* elsewhere, which manifest as *cliffs* in HRC at cache size intervals:

$$\{[SD(i \times \frac{T_{max}}{k}),\ SD((i+1) \times \frac{T_{max}}{k})], \quad \forall i \in \mathcal{I}\}$$

and *plateaus* elsewhere in the C domain, following Eq. (1).

3.3.2 Selecting the IRM Distribution.

g specifies an item frequency distribution from which the algorithm selects items directly and adds to the trace. The choice of g determines the item-frequency distribution (i.e., popularity). P_{IRM}, in turn, decides the fraction of these arrivals in the generated trace.

We note that f is a finite distribution, generating IRDs between 0 and a maximum value T_{max}. Fitting P_{IRM} and g involves examining the IRD distribution for values beyond T_{max}. The final IRD distribution is a merge of IRDs of dependent arrivals and independent arrivals, with the latter contributing to the tail.

3.3.3 Exploring Parameter Space for Desired Trace Behavior.

In practice, users don't need to understand the underlying fitting model. 2DIO provides an interactive visualization tool to assist parameter tuning. Users can (1) select a cache algorithm from the built-in cachesim library, (2) start with intuitive (or default) inputs, then (3) drag a slider or type in a number to change any single parameter and immediately observe the HRC recomputed on the fly.

While simulating the HRC on the fly can be resource-intensive, using a small trace footprint M and length N (e.g., 100, 10000, respectively) during this process minimizes overhead. Once tuned, parameters are reusable at various scales without compromising fidelity (see Sec. 5.3).

4 Implementation

2DIO package[5] includes a CLI tool `trace-gen` written in C++ for trace generation, and a Python library for cache simulation and analysis. `trace-gen` takes as input the footprint M, trace length N, and *trace profile* $\theta = \langle P_{\text{IRM}},\ g,\ f \rangle$.

Since we focus on block-level HRCs, all access units are assumed uniform, which is typical in storage cache evaluation.

This section details the implementation of the IRD and IRM samplers, introduces several default *trace profiles*, and provides example commands for generating traces with target cache behaviors.

4.1 Sampling Dependent Arrivals

`trace-gen` accepts f input through the **fgen** interface. We have preconfigured six default *trace profiles* for the toolkit. These generalize several real-world IRD patterns, labeled $\hat{f}_a$ through $\hat{f}_g$ (see Appendix A Table 6). Their corresponding HRCs and IRD distributions are visualized in Fig. 11 of the same appendix.

Auto-tuned IRD sample space. f is fitted to an auto-tuned k-binned sample space $\mathcal{S}$. The IRD sampler in Algorithm 2 selects a bin i with probability $f(i)$ and samples an IRD uniformly from that bin.

To ensure $\mathcal{S}$ matches the empirically measured IRD distribution from target trace π_s, T_{max} is auto-tuned based on M such that the mean of drawn IRD samples equals M. It follows that

$$M = \sum_{i=1}^{k} b_i \cdot f(i),$$

where b_i is the midpoint of bin i:

$$b_i = \frac{(2i-1)}{2} \times \frac{T_{max}}{k}.$$

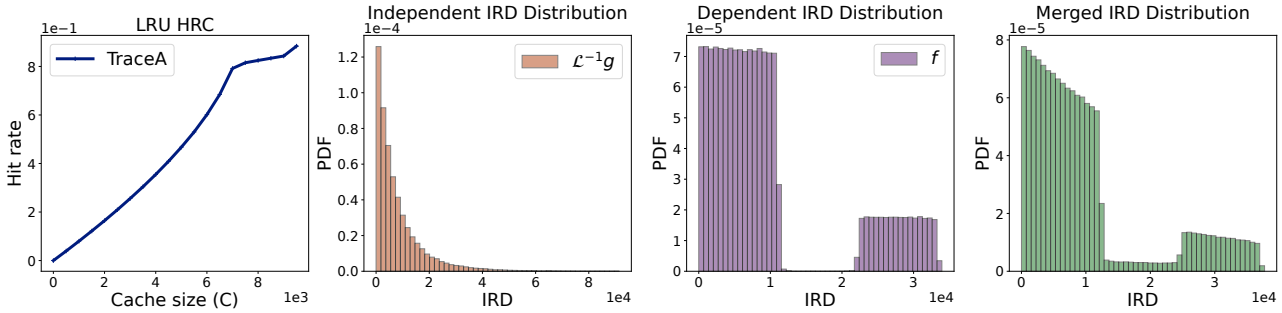
Hence T_{max} is solved by:

$$T_{max} = \frac{2Mk}{\sum_{i=1}^{k}(2i-1) \cdot f(i)}.$$

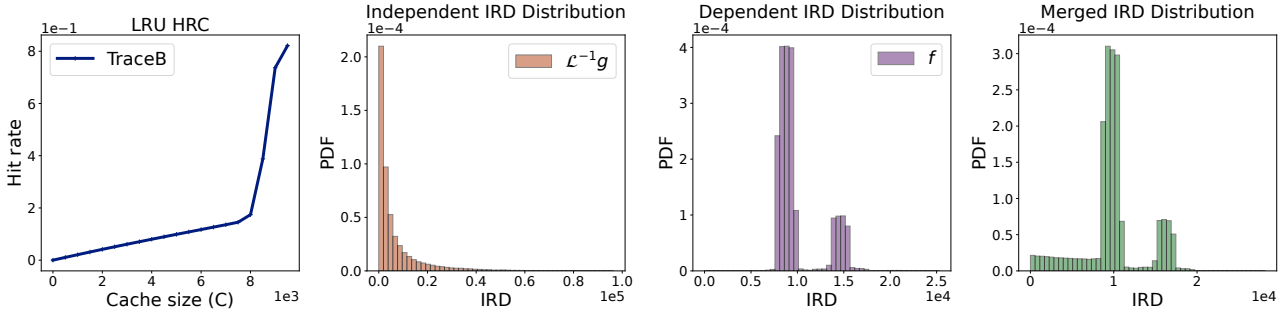
4.2 Sampling Independent Arrivals

The item-frequency distribution g for independent arrivals can follow Zipf, Pareto, Normal, or Uniform distributions over a sample space $\mathcal{U}$. `trace-gen` accepts the IRM type as a string input, defaulting to "zipf" with $\alpha = 1.2$ if unspecified.

[5]Available at https://github.com/Effygal/trace-gen; artifact also on https://zenodo.org/records/17202588.



(a) TraceA: $P_{IRM} = 0.1$, $g = \text{Zipf}(1.2)$, f fitted empirically to a simple 3-class IRD distribution



(b) TraceB: $P_{IRM} = 0.1$, $g = \text{Pareto}(2.5, 1)$, f fitted empirically to a 15-class IRD distribution

Figure 7. HRCs for TraceA and TraceB, with separate visualizations of IRD histograms for dependent, independent, and merged arrivals. $\mathcal{L}^{-1}g$ denotes the inverse Laplace transform of g, mapping its frequency-domain representation to the IRD domain.

For each specified IRM type, the associated parameters can be individually configured, generating a PMF g as shown in Table 2. The IRM sampler selects an item i from $\mathcal{U}$ with probability $g(i)$ following the corresponding PMF.

Table 2. `trace-gen` supported IRM types and corresponding PMF expressions.

IRM Type	Configurable	PMF
Zipfian	α	$g(i) = \left(\frac{1}{i}\right)^{\alpha}$
Pareto	α, x_m	$g(i) = \left(\frac{x_m}{i}\right)^{\alpha}$
Normal	μ, σ	$g(i) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(i-\mu)^2}{2\sigma^2}\right)$
Uniform	$a = 0$, $b = M - 1$	$g(i) = \frac{1}{M}$
Empirical	counts n_i	$g(i) = \frac{n_i}{\sum_j n_j}$

4.3 2D Generation Command

To generate TraceA with $f = \hat{f}_b$ and default $g = \text{Zipf}(1.2)$, use:

```
trace-gen -m 10000 -n 1000000 -f b -p 0.1
```

The resulting HRC and IRDs for independent, dependent, and merged arrivals are visualized in Fig. 7(a). To generate TraceB with an explicitly specified $f = \text{Pareto}(2.5, 1)$, use:

```
trace-gen -m 10000 -n 1000000 -f fgen
    :15:0.01:1,3,5,9 -g pareto:2.5,1 -p
    0.1
```

See Fig. 7(b) for the same visualization. In both traces, 90% of arrivals are sampled from f and only 10% from g, yielding minor independent influence. As a result, both traces exhibit highly *non-concave* HRCs.

One can ingest an I/O size distribution to vary request sizes if specified. E.g., `-sizedist 1,1,1:1,3,4` means equal chances of 1-, 3-, or 4-block requests. However, doing so may affect the carefully crafted IRD distribution. See Sec. 5.4 of this limitation.

5 Evaluation

This section evaluates 2DIO in terms of fidelity, configurability, and scalability. Evaluations involving real traces are conducted on the same subset described in Sec. 2.2. All cache simulations are performed using our built-in cachesim library.

Because the LRU hit ratio is entirely determined by recency [7, 20], we use LRU HRCs to assess how accurately our

method produces/reproduces recency which has not been achieved by existing generators.

All simulations are run on Ubuntu 24.04 LTS with AMD Ryzen 5 7600 6-Core Processor and 62 GB RAM. GAN hyperparameter searching and training jobs are run on an HPC node with 1 NVIDIA Tesla V100-SXM2 GPU, 2 CPU cores, and 8 GB of allocated memory.

5.1 Reproducing Real Trace with Succinct Parameters

Fidelity is assessed through HRC accuracy at block granularity. To test this we calibrate the profile θ for each trace, aiming to regenerate synthetic ones that produce similar HRCs—see Table 3, all of which are expressed with less than 10 numeric values. We then regenerate each on a small footprint $M = 100$ and length $N = 10$k except one[6], effectively reducing cache simulation costs.

Table 3. Parsimonious trace profiles for "counterfeiting" each real-world trace. **fgen** follows Eq. (3).

Trace ID	P_{IRM}	g	f
w11	1.0	Zipf(1.3)	None
w24	0.45	Zipf(1.2)	**fgen**$(30, [1, 2], 5e-3)$
w44	0.0	None	**fgen**$(30, [9, 13, 17, 19], 2.5e-2)$
w82	0.2	Zipf(1.2)	**fgen**$(100, [12, 13, 19], 1e-3)$
v521	0.0	None	**fgen**$(100, [2], 2e-3)$
v538	0.1	Zipf(1.2)	**fgen**$(40, [3, 4], 5e-3)$
v766	0.0	None	**fgen**$(40, [0, 5], 5.7e-3)$
v827	0.2	Zipf(1.2)	**fgen**$(60, [0, 13], 5e-3)$

Fig. 8 compares the HRCs of each 2DIO-generated trace (orange dashed) with their corresponding originals (blue solid). All relative *cliff* and *plateau* positions are preserved at cache sizes normalized to footprint M, with only minor deviations.

Comparison to Prior Work. TraceRaR [19] and generative adversarial networks (GANs) [43] represent state-of-the-art trace synthesize methods that are optimized for I/O replay-accuracy. Here we evaluate whether such optimizations also preserve HRC fidelity.

TraceRaR. TraceRaR [19] extends a given trace while preserving request rates, read/write ratios, and offset characteristics for effective scale-up replaying.

We used TraceRaR to extend each trace to twice its original length. In the synthesized output, the first half is identical to the original trace and therefore yields the same HRC. The second half, however, appears to be IRM; appending this segment to the real trace disrupts its recency. Consequently, the green dashed HRCs in Fig. 8 diverge markedly from the original.

LLGAN. LLGAN [43] utilizes a one-layer long short-term memory (LSTM) architecture for both the generator (G) and discriminator (D), and have both optimized against cross-entropy losses.

Since the original implementation is not publicly available and hyperparameter tuning is workload-specific, we reproduced the paper's proposed design[7]. Each trace is trained on two dimensions: `[LBA, Length]`[8]. We run several rounds of 50-trial hyperparameter searches with Optuna[2], looking for combinations where the D-loss settles around 0.5 and the G-loss is minimized, which often indicates a well-trained GAN. Table 4 summarizes the optimized hyperparameters for each trace, including random seeds used for reproducibility.

Sanity check. Before drawing any conclusion on HRC accuracy, we first validate the LLGAN-synthesized traces against the evaluation metric used in the original paper—the maximum mean discrepancy[9] (MMD^2), computed as the joint distributional difference over LBAs and lengths. The last two columns of Table 4 report the training D-loss and resulting MMD^2; $MMD^2 \to 0$ indicates the joint distribution of synthesized LBAs and lengths are close to the original trace—see Appendix B Fig. 12 for explicit visual comparisons.

The red dashed curves appear in Fig. 8 show LLGAN-generated HRCs. Notably, accurately reproducing LBA and length distributions does not imply HRC-fidelity. E.g., the synthetic w44 yields the closest HRC to the original despite a high MMD^2, whilst the synthetic w11 has the lowest MMD^2 but produces a clearly mismatched HRC.

5.2 Creating a Full Spectrum of "What-if" Traces

Typical microbenchmarks [4, 6, 10, 17, 18, 22] model workloads primarily through a few coarse knobs—read/write ratio, object size, or broader "workload personalities" such as database, file, or web servers. The cache sees whatever spatio-temporal locality naturally arises from those aggregate knobs, making the workloads effectively cache-agnostic at block level. Few of them [4, 17, 18] allow IRM block frequency specifications, e.g., Zipf, Pareto, or customized Zoned distributions. Table 5 summarizes the random types supported by each tool, mostly limited to independent access patterns.[10]

A key advantage of 2DIO is its ability to tune the full spectrum of parameters, yielding a continuum of workload

[6] w44 may require $M \geq 10$k and $N \geq 1$m for a high-resolution HRC-reproduction.

[7] Available at https://github.com/Effygal/gan-io

[8] Only logical block addresses (LBAs) and I/O sizes (lengths) are relevant to cache performance.

[9] Computed with an RBF kernel and median bandwidth σ.

[10] fio and 2DIO support heterogeneous request sizes (e.g., multi-block), inducing sequential access and triggering scan-like cache behavior, though scan locations remain intractable

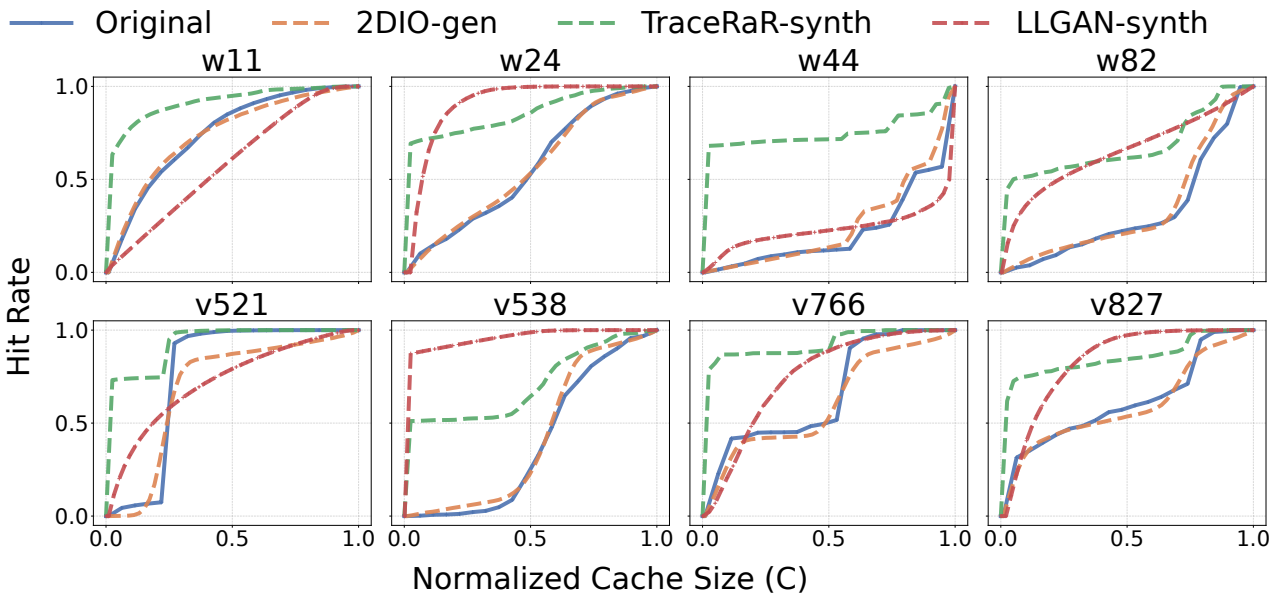


Figure 8. 2DIO reproducing real-world trace HRCs compared to TraceRaR and LLGAN synthesized traces.

Table 4. Optuna-optimized hyperparameters for LLGAN synthesis with resulted D-loss and MMD^2.

Trace	Rand Seed	Hidden dim	Latent dim	Batch size	Sequence len	Epochs	G-rate	D-rate	G-updates	D-updates	D-loss	MMD^2
w11	77	126	10	64	12	30	0.000132	0.000454	1	2	0.5411	0.005128
w24	77	108	10	128	12	30	0.000090	0.000397	2	3	0.5308	0.071148
w82	42	121	10	128	12	30	0.000075	0.000108	1	2	0.4856	0.047067
w44	42	100	16	128	12	30	0.000248	0.000139	3	2	0.5600	0.060406
v521	77	101	10	64	12	30	0.000099	0.000157	1	3	0.5079	0.118840
v538	42	102	10	64	12	30	0.000415	0.000281	1	3	0.5510	0.016633
v827	77	108	10	128	12	30	0.000090	0.000397	2	3	0.5099	0.080228
v766	77	111	10	64	12	30	0.000254	0.000141	1	3	0.4818	0.013860

Table 5. Random types supported by available tools. Only synthetic I/O generation is shown here, a tiny fraction of their full feature set.

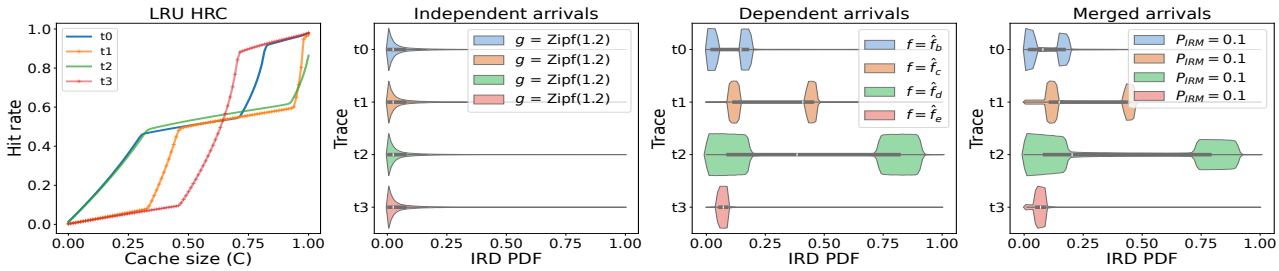
Tool	Independent						Dependent		Mixed
	zipf	pareto	zoned	normal	unif.	emp.	seq.	IRD	
IOzone [22]	✗	✗	✗	✗	✗	✗	✗	✗	✗
Bonnie++ [6]	✗	✗	✗	✗	✗	✗	✗	✗	✗
IOmeter [18]	✗	✗	✓	✗	✗	✗	✗	✗	✗
Sysbench [17]	✓	✓	✓	✗	✗	✗	✗	✗	✗
FIO [4]	✓	✓	✓	✓	✓	✗	✓	✗	✗
2DIO	✓	✓	✓	✓	✓	✓	✓	✓	✓

behaviors. To evaluate this, we use $\hat{f}_b$ to $\hat{f}_g$[11] (see Sec. 4.1) to generate 12 example traces t0–t11, each with footprint $M = 10k$ and length $N = 1m$. Fig. 9 shows a separated view of the IRDs for independent, dependent, and merged arrivals, alongside the resulting HRCs. Each sub-figure illustrates how modulating f, g, and P_{IRM} individually affects the simulated HRCs. Fig. 9(a) shows how shifting *spike* positions in f causes corresponding changes in *cliff* and *plateau* positions. With fixed $P_{IRM} = 0.1$, IRD *spikes* primarily dictate the HRC *cliff* while the IRM traffic has minor influence. Fig. 9(b) shows how switching IRM types g affects the HRC shapes. Since 90% of arrivals are independent, $\hat{f}_f$ incurs only minor influence on the merged IRDs, yielding exponential-like merged IRDs and concave HRCs across all traces. Fig. 9(c) shows the effects of adjusting P_{IRM}, where t7–t11 are generated with fixed $g = \text{Zipf}(1.2)$ and $f = \hat{f}g$; gradually increasing P_{IRM} from 0.1 to 0.9 results in progressively higher HRC concavity.

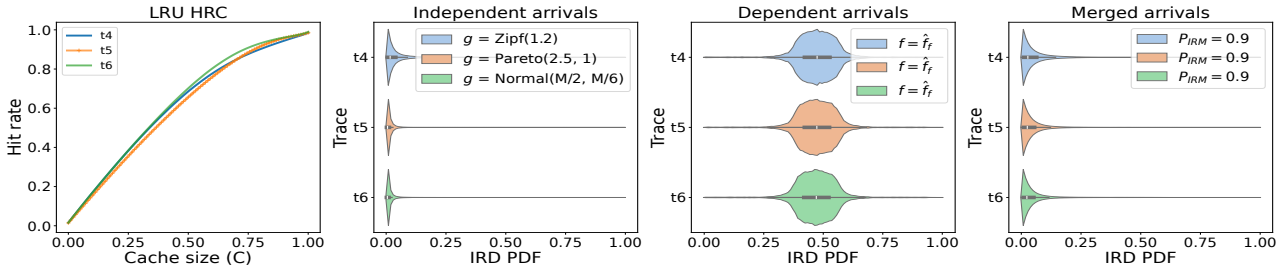
[11] $\hat{f}g = \textbf{fgen}((54, [5, 11, 12, 13, 14, 17, 30, 50], 1e-2))$; *spike* values are manually re-distributed.

5.3 Fidelity-Persistant Up- and Down-scaling

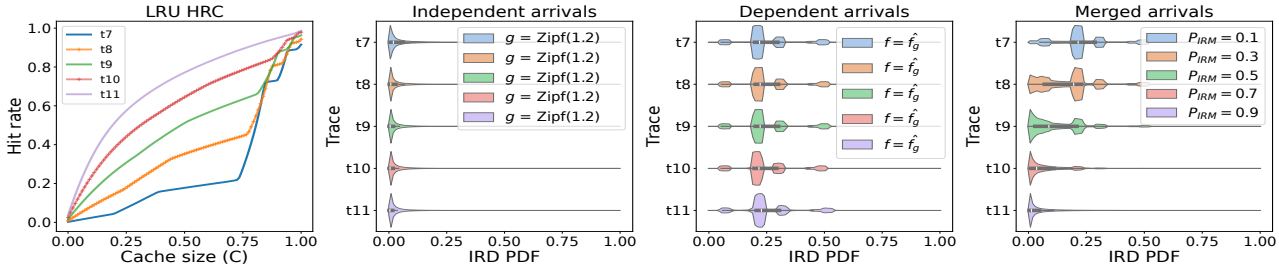
For this evaluation we regenerate each trace at various scales under fixed θ (see Sec. 5.1 Table 3), assessing the HRC accuracy in terms of mean absolute error (MAE) at each scale against the original.



(a) Varying f to configure HRC *cliff* and *plateau* positions



(b) Varying g to configure concave HRC shapes



(c) Varying P_{IRM} to configure HRC concavity level

Figure 9. Effect of varying each parameter on the resulting HRCs, shown with example 2DIO-generated traces for M = 10k and N = 1m. The three violin plots, from left to right, depict: (1) the IRD PDF under IRM's independent arrivals; (2) the IRD PDF from dependent arrivals sampled via the input f; and (3) the combined IRD PDF obtained by merging both processes.

Scaling N and M simultaneously. We scale M down from 10k to 100 and N from 1m to 10k, maintaining a fixed N/M ratio of 100. Fig. 10(a) presents the results. The HRCs generated by 2DIO at three scales are compared with the original one. MAEs are consistently at around 0.03 to 0.05. While the resolution of *cliffs* is "smoothed out" at the smallest scale (M = 100, N = 10k), the overall cache behavior persists.

Scaling footprint M. With N fixed at 100k, varying M from 10k to 100 produces results in Fig. 10(b) where MAEs are found to be between 0.02 and 0.05.

Scaling trace length N. Here M is fixed at 1k. Fig. 10(c) shows the HRCs and MAEs as N varies from 10k to 1m; MAEs are around 0.04.

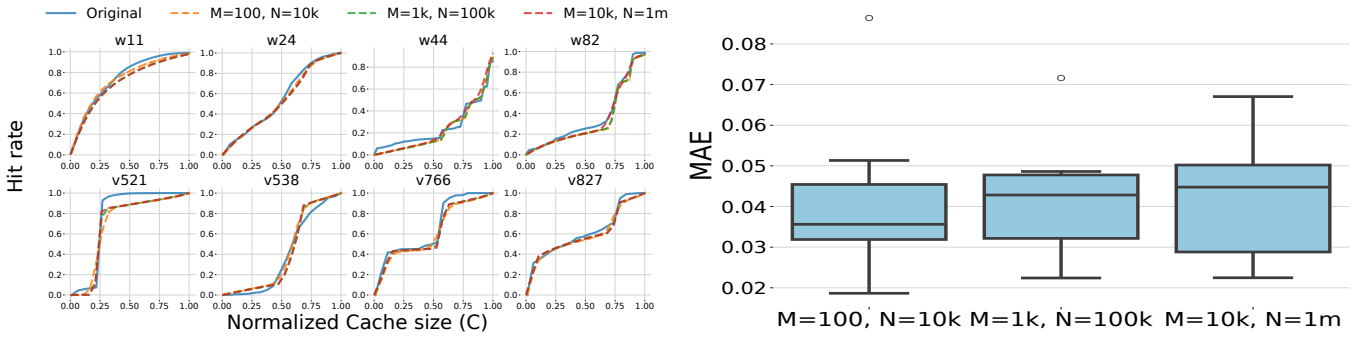
Observe that MAE does not necessarily improve/worsen with larger/smaller scales. Accuracy primarily depends on how well the θ parameters are tuned at the initial scales.

5.4 Limitations

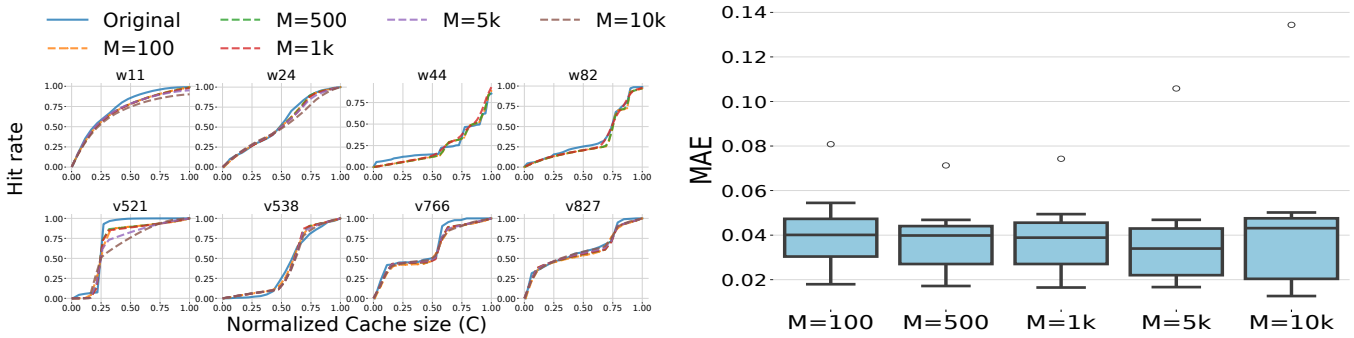
Traces generated by 2DIO follow the SPC format [32] and can be replayed on any storage systems. Users can specify a read/write ratio and a request size distribution. However, specifying a request size distribution that mixes single- and multi-block references effectively introduces sequential accesses at arbitrary locations, which may distort the IRD sample space and render the crafted *spikes* intractable.

6 Related Work

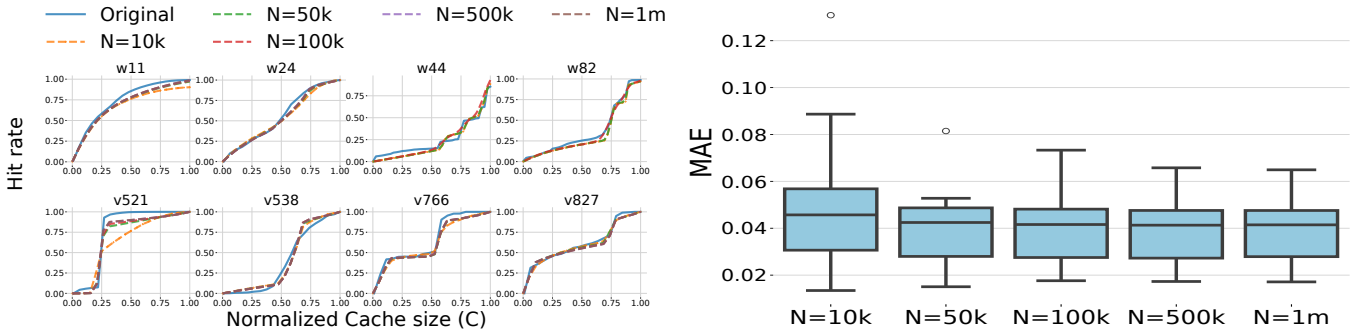
Re-scaled simulations. A closely related field are scaling down simulations proposed by Waldspurger et al. [34, 35], which approximates the LRU HRC by sampling requests from a given trace while preserve its underlying stack distance distribution.



(a) Scaling M and N with fixed N/M



(b) Scaling M with fixed N



(c) Scaling N with fixed M

Figure 10. Scalability test results. HRC accuracy under various scales of M and N over metric MAE.

The fundamental difference between these works and 2DIO is that they translate a trace to an HRC; 2DIO does the opposite, translating an HRC to trace(s). The similarities between the two are due to both relying on prior cache modeling works going back decades (Mattson et al., 1970 [20]; Denning, 1968 [7]).

Scaling up synthetic traces enhances debugging and performance testing by extending trace length while preserving key distributions, as in TraceRaR [19]. More recently, scaling up has enabled distributed storage evaluations on single nodes or small clusters by combining down-sampled traces from multiple storage units [23], creating representative or counterfactual traces based on disk capacity and placement policies.

Workload Synthesis for CDNs. There is a stream of workload generation tools [26, 27, 30] used in production CDNs. The most recent JEDI [27] produces a synthetic trace that simultaneously matches the object-level properties (object size distribution, popularity distribution, request size distribtuion) and the cache-level properties (object- and byte-level HRCs) of the original trace. Their evaluations report small HRC simulation errors on workload regenerations across various policies, while all reported HRCs appear to indicate IRM-like workloads.

7 Conclusion

2DIO offers a practical way to generate synthetic traces that are both realistic and predictable. By combining short-term recency and long-term frequency characteristics in a concise parameter triplet, it produces traces with complex, *non-concave* hit ratio curves, including performance *cliffs* and *plateaus* seen in real workloads. The parameterization is succinct enough to be explored exhaustively in experiments, yet highly scalable to adapt to various systems under evaluation without altering cache behavior. It is lightweight and easily integrates into existing benchmarking tools, enabling a standardized, reproducible, and shareable platform for cross-system benchmarking.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Bhuvan Urgaonkar, for their extensive and constructive feedback. This work was made possible thanks in part to support from Red Hat, VMware/Broadcom and NSF award CNS-1910327.

References

[1] Alfred V Aho, Peter J Denning, and Jeffrey D Ullman. 1971. Principles of optimal page replacement. *Journal of the ACM (JACM)* 18, 1 (1971), 80–93.

[2] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. 2019. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*. 2623–2631.

[3] Alibaba. [n. d.]. Alibaba/block-traces. https://github.com/alibaba/block-traces

[4] J. Axboe. 2003. GitHub - axboe/fio: Flexible I/O Tester — github.com. https://github.com/axboe/fio. [Accessed 08-07-2024].

[5] E. G. Coffman and Peter J. Denning. 1973. *Operating systems theory*. Prentice-Hall, Englewood Cliffs, N.J.

[6] R. Coker. 2003. Bonnie++ now at 1.03e (last version before 2.0)! — coker.com.au. https://www.coker.com.au/bonnie++/. [Accessed 08-07-2024].

[7] Peter J. Denning. 1968. The working set model for program behavior. *Commun. ACM* 11, 5 (May 1968), 323–333. doi:10.1145/363095.363141

[8] Peter J Denning and Stuart C Schwartz. 1972. Properties of the working-set model. *Commun. ACM* 15, 3 (1972), 191–198.

[9] David Eklov and Erik Hagersten. 2010. StatStack: Efficient modeling of LRU caches. In *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*. IEEE, 55–65.

[10] Filebench. [n. d.]. Filebench/filebench: File system and storage benchmark that uses a custom language to generate a large variety of workloads. https://github.com/filebench/filebench

[11] Hao Che, Ye Tung, and Zhijun Wang. 2002. Hierarchical Web caching systems: modeling, design and experimental results. *IEEE Journal on Selected Areas in Communications* 20, 7 (Sept. 2002), 1305–1314. doi:10.1109/JSAC.2002.801752

[12] Gerhard Hasslinger, Konstantinos Ntougias, Frank Hasslinger, and Oliver Hohlfeld. 2023. Scope and Accuracy of Analytic and Approximate Results for FIFO, Clock-Based and LRU Caching Performance. *Future Internet* 15, 3 (Feb. 2023), 91. doi:10.3390/fi15030091

[13] Xiameng Hu, Xiaolin Wang, Yechen Li, Lan Zhou, Yingwei Luo, Chen Ding, Song Jiang, and Zhenlin Wang. 2015. {LAMA}: Optimized locality-aware memory allocation for key-value cache. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. 57–69.

[14] Xiameng Hu, Xiaolin Wang, Lan Zhou, Yingwei Luo, Chen Ding, and Zhenlin Wang. 2016. Kinetic Modeling of Data Eviction in Cache. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association, Denver, CO, 351–364.

[15] IRCache. 2000. The First Semi-Annual Web Caching Cache-off. http://www.ircache.net/n01 (archived at https://web.archive.org/web/*/http://www.ircache.net/n01).

[16] Minji Kang, Soyee Choi, Gihwan Oh, and Sang-Won Lee. 2020. 2r: Efficiently isolating cold pages in flash storages. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2004–2017.

[17] Alexey Kopytov. 2004. Sysbench: a system performance benchmark. *http://sysbench. sourceforge. net/* (2004).

[18] David D Levine. 1998. Iometer user's guide. *Intel Server Architecture Lab* 40 (1998).

[19] Bingzhe Li, Farnaz Toussi, Clark Anderson, David J Lilja, and David HC Du. 2017. Tracerar: An i/o performance evaluation tool for replaying, analyzing, and regenerating traces. In *2017 International Conference on Networking, Architecture, and Storage (NAS)*. IEEE, 1–10.

[20] R.L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. 1970. Evaluation techniques for storage hierarchies. *IBM Systems Journal* 9, 2 (1970), 78–117. doi:10.1147/sj.92.0078

[21] Qingpeng Niu, James Dinan, Qingda Lu, and Ponnuswamy Sadayappan. 2012. PARDA: A fast parallel reuse distance analysis algorithm. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*. IEEE, 1284–1294.

[22] William D Norcott. 2003. Iozone filesystem benchmark. *http://www.iozone. org/* (2003).

[23] Phitchaya Mangpo Phothilimthana, Saurabh Kadekodi, Soroush Ghodrati, Selene Moon, and Martin Maas. 2024. Thesios: Synthesizing Accurate Counterfactual I/O Traces from I/O Samples. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 1016–1032.

[24] Roman Pletka, Ioannis Koltsidas, Nikolas Ioannou, Saša Tomić, Nikolaos Papandreou, Thomas Parnell, Haralampos Pozidis, Aaron Fry, and Tim Fisher. 2018. Management of next-generation NAND flash to achieve enterprise-level endurance and latency targets. *ACM Transactions on Storage (TOS)* 14, 4 (2018), 1–25.

[25] Mendel Rosenblum and John K. Ousterhout. 1991. The design and implementation of a log-structured file system. In *13th ACM symposium on Operating systems principles*. ACM, Pacific Grove, California, United States, 1–15.

[26] Anirudh Sabnis and Ramesh K Sitaraman. 2021. TRAGEN: a synthetic trace generator for realistic cache simulations. In *Proceedings of the 21st ACM Internet Measurement Conference*. 366–379.

[27] Anirudh Sabnis and Ramesh K Sitaraman. 2022. JEDI: model-driven trace generation for cache simulations. In *Proceedings of the 22nd ACM Internet Measurement Conference*. 679–693.

[28] Rathijit Sen and David A Wood. 2013. Reuse-based online models for caches. In *Proceedings of the ACM SIGMETRICS/international conference on Measurement and modeling of computer systems*. 279–292.

[29] Radu Stoica, Roman Pletka, Nikolas Ioannou, Nikolaos Papandreou, Sasa Tomic, and Haris Pozidis. 2019. Understanding the design trade-offs of hybrid flash controllers. In *2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 152–164.

[30] Aditya Sundarrajan, Mingdong Feng, Mangesh Kasbekar, and Ramesh K Sitaraman. 2017. Footprint descriptors: Theory and practice of cache provisioning in a global cdn. In *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies*. 55–67.

[31] Jian Tan, Tieying Zhang, Feifei Li, Jie Chen, Qixing Zheng, Ping Zhang, Honglin Qiao, Yue Shi, Wei Cao, and Rui Zhang. 2019. ibtune: Individualized buffer tuning for large-scale cloud databases. *Proceedings of*

the VLDB Endowment 12, 10 (2019), 1221–1234.

[32] Storage Performance Council University of Massachusetts, Amherst. n.d.. SPC Traces: Storage Performance Council Traces. https://skulddata.cs.umass.edu/traces/storage/SPC-Traces.pdf. Accessed: 2025-01-02.

[33] Jacob van den Berg and D Toswley. 1993. Properties of the miss ratio for a 2-level storage model with LRU or FIFO replacement strategy and independent references. *IEEE Trans. Comput.* 42, 4 (1993), 508–512.

[34] Carl Waldspurger, Trausti Saemundsson, Irfan Ahmad, and Nohhyun Park. 2017. Cache Modeling and Optimization using Miniature Simulations. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 487–498.

[35] Carl A. Waldspurger, Nohhyun Park, Alexander Garthwaite, and Irfan Ahmad. 2015. Efficient MRC Construction with SHARDS. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*. USENIX Association, Santa Clara, CA, 95–110.

[36] Hua Wang, Jiawei Zhang, Ping Huang, Xinbo Yi, Bin Cheng, and Ke Zhou. 2020. Cache what you need to cache: Reducing write traffic in cloud cache via "one-time-access-exclusion" policy. *ACM Transactions on Storage (TOS)* 16, 3 (2020), 1–24.

[37] Qiuping Wang, Jinhong Li, Patrick PC Lee, Tao Ouyang, Chao Shi, and Lilong Huang. 2022. Separating data via block invalidation time inference for write amplification reduction in {Log-Structured} storage. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*. 429–444.

[38] D.L. Willick, D.L. Eager, and R.B. Bunt. 1993. Disk cache replacement policies for network fileservers. In *[1993] Proceedings. The 13th International Conference on Distributed Computing Systems*. IEEE Comput. Soc. Press, Pittsburgh, PA, USA, 2–11. doi:10.1109/ICDCS.1993.287729

[39] Jake Wires, Stephen Ingram, Zachary Drudi, Nicholas JA Harvey, and Andrew Warfield. 2014. Characterizing storage workloads with counter stacks. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*. 335–349.

[40] Jake Wires, Stephen Ingram, Zachary Drudi, Nicholas J. A. Harvey, and Andrew Warfield. 2014. Characterizing Storage Workloads with Counter Stacks. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 335–349.

[41] Xiaoya Xiang, Bin Bao, Chen Ding, and Yaoqing Gao. 2011. Linear-time modeling of program working set in shared cache. In *2011 International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 350–360.

[42] Zhu Yuan, Xueqiang Lv, Ping Xie, Haojie Ge, and Xindong You. 2024. CSEA: a fine-grained framework of climate-season-based energy-aware in cloud storage systems. *Comput. J.* 67, 2 (2024), 423–436.

[43] Heyu Zhang, Zhen Yang, Yulai Xie, Yafeng Wu, Jiakun Li, Dan Feng, Avani Wildani, and Darrell Long. 2024. Accurate Generation of I/O Workloads Using Generative Adversarial Networks. In *2024 International Conference on Networking, Architecture and Storage (NAS)*. IEEE, 1–9.

A 2DIO Built-in Trace Profiles

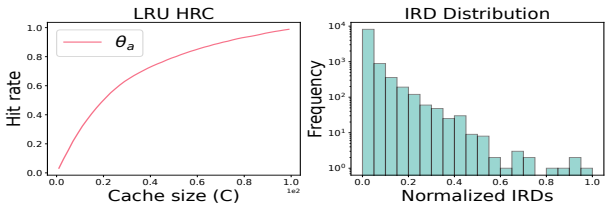
Table 6 lists the default trace profiles introduced in Sec. 4.1, with **fgen** defined by Eq. (3). Fig. 11 visualizes the HRC and IRD distributions for each profile, presenting a set of canonical workload behaviors that reflect commonly observed real-world patterns.

Table 6. 2DIO default trace profiles.

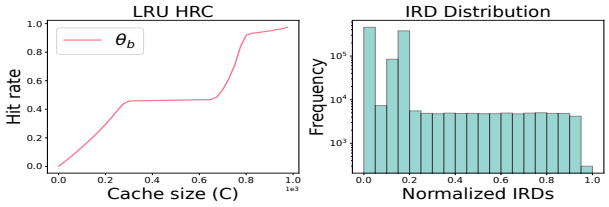
	P_{IRM}	g	f
θ_a	1.0	Zipf(3.0)	$\hat{f}_a$ = None
θ_b	0.0	None	$\hat{f}_b$ = **fgen**(20, [0, 3], $5e-3$)
θ_c	0.0	None	$\hat{f}_c$ = **fgen**(20, [2, 9], $5e-3$)
θ_d	0.0	None	$\hat{f}_d$ = **fgen**(5, [0, 4], $1e-2$)
θ_e	0.0	None	$\hat{f}_e$ = **fgen**(20, [1], $5e-3$)
θ_f	0.0	None	$\hat{f}_f$ = **fgen**(5, [2], $5e-3$)

B LLGAN Synthetic Traces Validation

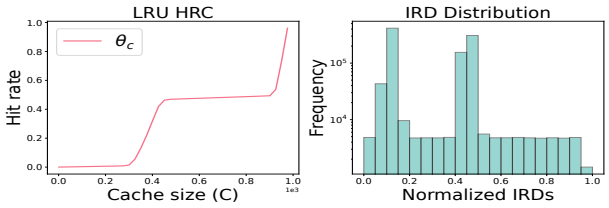
Fig. 12 plots the kernel density estimation (KDE) of LBA and length fidelity for LLGAN-synthesized traces against the originals as a sanity check for Sec. 5.1. Synthetic w11 matches the originals most closely, in line with its lowest MMD^2 seen in Table 4, whereas synthetic w44 shows a clear deviation (median $MMD^2 \approx 0.06$), yet produces the highest HRC fit.



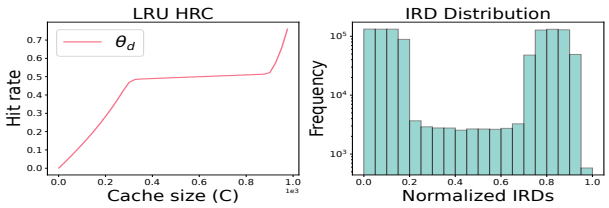
(a) $\theta_a = \langle P_{IRM} = 1.0, g = \text{Zipf}(3.0), f = \text{None} \rangle$



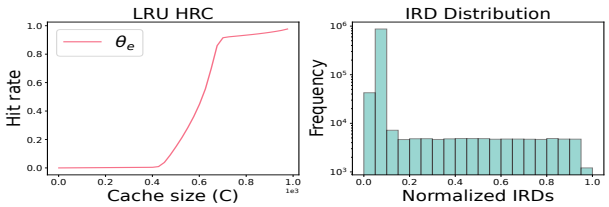
(b) $\theta_b = \langle P_{IRM} = 0.0, g = \text{None}, f = \hat{f}_b \rangle$



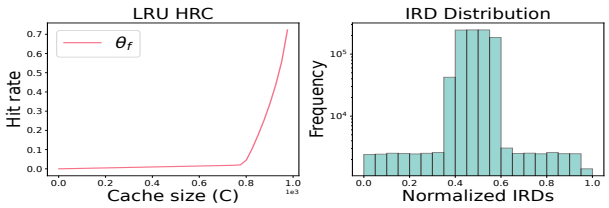
(c) $\theta_c = \langle P_{IRM} = 0.0, g = \text{None}, f = \hat{f}_c \rangle$



(d) $\theta_d = \langle P_{IRM} = 0.0, g = \text{None}, f = \hat{f}_d \rangle$



(e) $\theta_e = \langle P_{IRM} = 0.0, g = \text{None}, f = \hat{f}_e \rangle$



(f) $\theta_f = \langle P_{IRM} = 0.0, g = \text{None}, f = \hat{f}_f \rangle$

Figure 11. HRCs and IRD distributions for the 2DIO default trace profiles θ_a–θ_f.

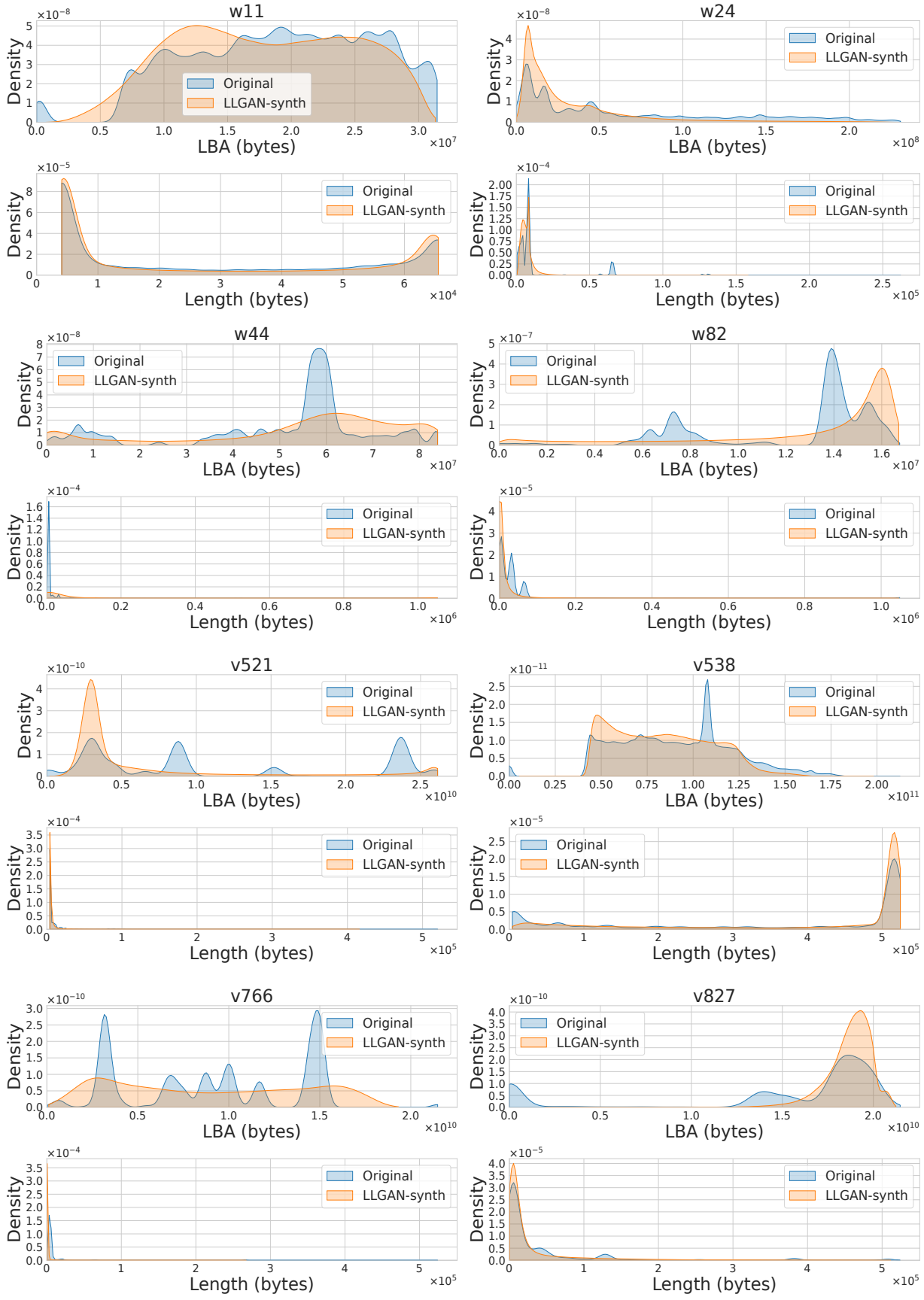


Figure 12. KDEs of LBAs and lengths, showing distributional similarity between LLGAN-synthesized traces vs. originals.