Writeback Modeling: Theory and Application to Zipfian Workloads

Wesley Smith
wsmith6@cs.rochester.edu
University of Rochester
United States

Daniel Byrne
djbyrne@mtu.edu
Michigan Technological University
United States

Chen Ding
cding@cs.rochester.edu
University of Rochester
United States

ABSTRACT

As per-core CPU performance plateaus and data-bound applications like graph analytics and key-value stores become more prevalent, understanding memory performance is more important than ever. Many existing techniques to predict and measure cache performance on a given workload involve either static analysis or tracing, but programs like key-value stores can easily have billions of memory accesses in a trace and have access patterns driven by non-statically observable phenomena such as user behavior. Past analytical solutions focus on modeling cache hits, but the rise of non-volatile memory (NVM) like Intel's Optane with asymmetric read/write latencies, bandwidths, and power consumption means that writes and writebacks are now critical performance considerations as well, especially in the context of large-scale software caches. We introduce two novel analytical cache writeback models that function for workloads with general frequency distributions; in addition we provide closed-form instantiations for Zipfian workloads, one of the most ubiquitous frequency distribution types in data-bound applications. The models have different use cases and asymptotic runtimes, making them suited for use in different circumstances, but both are fully analytical; cache writeback statistics are computed with no tracing or sampling required. We demonstrate that these models are extremely accurate and fast: the first model, for infinitely large level-two (L2) software cache, averaged 5.0% relative error from ground truth and achieved a minimum speedup over a state-of-the-art trace analysis technique (AET) of 515x to generate writeback information for a single cache size. The second model, which is fully general with respect to L1 and L2 sizes but slower, averaged 3.0% relative error from ground truth and achieved a minimum speedup over AET of 105x for a single cache size.

CCS CONCEPTS

• **Computing methodologies** → **Modeling methodologies**; • **Information systems** → **Hierarchical storage management**.

KEYWORDS

writebacks, modeling, Zipfian distributions, locality, hierarchical memory

1 INTRODUCTION

At its core, locality analysis seeks to rigorously understand the interaction between a program and a memory hierarchy. This relationship is multifaceted, and understanding program memory access patterns as well as caching system structure is essential to effective analysis.

This work explores the interactions between programs with Zipfian access patterns and caching systems, specifically through the lens of understanding writes and writebacks. This focus is of growing importance with the advent of memory technologies such as Intel's Optane, which is of growing interest in the memory community [25, 30]. Optane has asymmetric read/write behavior: its read bandwidth is around 3x that of its write bandwidth, and writes incur substantially more latency and power consumption [19]. We introduce probabilistic analytical models of cache behavior that are accurate and effective at predicting caching events while being closed form: as prior work tends to rely on tracing and sampling [10, 17] or focuses on modeling hits [7, 14, 21, 27], these models represent not only a substantial theoretic contribution but also a significant improvement in asymptotic cost and performance. Besides cost, another problem with trace analysis is "...the trace reflects only a specific workload, and there is always the question of whether the results generalize to other systems or load conditions..." [13]. There exist tools that use static analysis for cache performance prediction [9], but these require the program memory access pattern to be statically inferrable, which is often not the case for things like key-value stores and graph applications. As such, our modeling work represents a domain extension over these static analysis techniques.

Many existing locality models and techniques [9, 10, 34, 39] focus on *reuse*, characterizing the intervals between consecutive accesses to the same memory location (hardware caching) or object (software caching). This is especially apt for programs whose memory access pattern is driven by static, source-level references. Another paradigm entirely is *frequency analysis*, built on the Independent Reference Model (IRM) [32], which views program memory accesses as sequences of independent samples to a frequency distribution.

Frequency analysis is the focus of this work.

One of the most important and ubiquitous types of frequency distribution is a Zipfian (power-law) distribution, which is found in many disparate areas of computer science [3, 26, 35, 40]. In the context of frequency analysis of programs, Zipfian distributions are commonly found in situations where memory access patterns are driven by things beyond static program references, such as user behavior in key-value stores or the structure of data in graph analytics.

In the background section we provide a brief overview of this work's parent project, CLUsivity. CLUsivity is a hybrid DRAM-NVM software caching system that seeks to jointly optimize key-value stores over both cache misses and writes. The main technique involves offline parameter-space exploration of a combinatorial number of system configurations, something made possible by the existence of fast, fully closed-form models that require no traces. In this way, the CLUsivity project provides both motivation for this modeling and a demonstration of its applicability and utility.

The rest of the paper is structured as follows: Section 2 contains background information and related work discussion. Section 3 contains the analytical models that are the core contributions of the paper, analyses of their runtimes, implementation details, and discussion of their limitations. Section 4 contains empirical results from validating our models against simulators and a state-of-the-art sampling technique. Section 5 contains discussion of how our work can be extended and strengthened as well as concluding remarks.

1.1 Main Contributions

The main contributions of this work are as follows:

- An *Infinite L2 Writeback Model* (**Proposition 1**), which introduces an analytical model of LRU cache writebacks for infinitely large L2, and **Theorem 1**, which instantiates the model for Zipfian access patterns and presents an optimized and closed-form result.
- A *General Writeback Model* (**Proposition 2**), which introduces a second, fully general analytical model of LRU cache writebacks, and **Theorem 2**, which instantiates the model for Zipfian access patterns as before.
- **Evaluation** of the performance and runtimes of our models against simulation and a state-of-the-art trace sampling technique.

2 BACKGROUND

2.1 Zipfian Access Patterns

One of the most ubiquitous types of memory access pattern is characterized by a *Zipfian* distribution, with the following probability density function:

$$p(i) = \frac{\omega(\alpha, m)}{i^\alpha} \tag{1}$$

where:

- i: the *rank* of the element in question
- m: the datasize (number of unique objects)
- α: the Zipfian distribution parameter
- $\omega(\alpha, m)$: a normalizing term
- $p(i)$: the probability of accessing element i

The parameter α dictates the gap in probability density between consecutively ranked items. Higher α means a larger gap, and thus better locality. $\alpha = 0$ characterizes a uniform distribution.

Such power-law distributions are some of the most fundamental and omnipresent mathematical objects in computer science, appearing everywhere from corpora analytics in natural language processing [26], to key-value store access patterns [40], to traffic routing at both the hardware (per-router traffic) level and the per-website user traffic level [3], to partitioning techniques for graph analytics [35]. This work's parent project, CLUsivity, builds a K-V store to run on top of hybrid DRAM/NVM caching systems. Zipf distributions in this context often have α values that "...fall in [0.9, 0.99] for daily scenarios, and in [1, 1.22] for extreme cases" [40]. Yang et al. [38] show that block I/O writes in storage systems follow Zipfian popularity distributions as well.

2.2 Analytical Miss Ratio Modeling

The literature on modeling the performance of LRU caches is extensive and dates back to the 1970s, with Ronald Fagin's seminal probabilistic counting approach. Fagin describes computing the *expected working-set size*, or expected number of unique elements in a length-x trace window, as follows [12]:

$$wss_{avg}(x) = fp(x) = \sum_{i=1}^{m} (1 - (1 - p(i))^x) \tag{2}$$

This expression has a straightforward intuitive explanation: $(1 - p(i))^x$ is the probability that element i does *not* appear in a given window of length x in the trace, so its complement is the probability that it appears one or more times. Summing this quantity over all data results in expected working-set size.

The working-set function concept is expanded on in Yuan et al. [39], who use the term *footprint*, or *fp(x)*. The Denning-HOTL conversion, discussed in the same paper, shows how *fp(x)* captures the performance of a trace on an LRU cache of size c:

$$mr(c) = \Delta fp(x)|_{fp(x)=c} \tag{3}$$

In words, the miss ratio for an LRU cache of size c is the finite difference of the footprint function *at the point where* $fp(x) = c$. The core ideas behind this relationship are visualized in Figure 1.

We can combine Equations 1 and 2 to get an approximation of the footprint function for a Zipfian probability distribution:

$$fp(x) = \sum_{i=1}^{m} 1 - \left(\frac{i^\alpha - \omega(m, \alpha)}{i^\alpha}\right)^x \tag{4}$$

Differentiating Equation 4 will give an approximate LRU miss ratio curve. However, since window size x ranges from 0 to n, trace length, and at each step we must sum over m, datasize, this approach is intractable for real traces. As such, a closed form solution

is desirable.

Jelenković [20] gave a closed form approximation for Zipfian distributions with $\alpha > 1$ in 1999, which is generalized for all values of α by Christian Berthet [5] in a 2017 publication. His model, although using different domain language, computes approximate $fp(x)$ (and by extension miss ratio) as follows:

$$fp(x) \approx m\left(1 - \frac{1}{\alpha} \cdot E_{1+\frac{1}{\alpha}}\left(\frac{x}{H_{m,\alpha} \cdot m^{\alpha}}\right)\right) \tag{5}$$

Throughout the rest of this paper we refer to Equation 5 as the Jelenković-Berthet approximation. Here $H_{m,\alpha}$ is the m_{th} generalized harmonic number, related to our normalizing term from before as follows:

$$\frac{1}{\omega(m,\alpha)} = \sum_{i=1}^{m} \frac{1}{i^{\alpha}} = H_{m,\alpha} \tag{6}$$

and $E_k(x)$ is the k_{th} generalized exponential integral:

$$E_k(x) = \int_1^{\infty} \frac{e^{-x \cdot t}}{t^k} dt \tag{7}$$

Equation 5 (miss ratio) is a required component in one of the models presented in the following sections: we implement it in Rust using the RGSL math library.

2.3 Average Eviction Time Cache Modeling

The Average Eviction Time (AET) model [17, 18] is an online trace based approach to computing cache performance information. AET first samples requests to build a reuse interval histogram. The histogram is then used to calculate the probability that reference x has reuse time greater than t, which is then related to a stack movement in an LRU queue. Note that AET and cache fill time are equivalent measures.

Given a reuse interval histogram we can solve for the average eviction time for a given cache size, c. It is realized by the following equation:

$$\int_0^{AET(c)} P(t)dt = c \tag{8}$$

where $P(t)$ is the probability that a reference has reuse interval greater than t. We obtain $P(t)$ from the reuse time histogram. Armed with the *average* eviction time for a cache of size c, we can now compute the miss ratio curve using the following equation:

$$mr(c) = P(AET(c)) \tag{9}$$

where $P(AET(c))$ is the probability that a reuse interval is greater than the average eviction time for cache of size c, $AET(c)$. Figure 1 explains this visually.

AET can be extended to calculate writeback ratio curves for infinite and finite L2 cache sizes. For infinite L2 caches, AET implements Chen's writeback model formula [10]. For finite L2 caches, AET implements the equations outlined in Section 3.3 for generalized modeling. Specifically, the conditions in equations 23 and 25 are met by keeping track of each backward reuse interval for each write interval. The additional complexity increases the accuracy of AET's writeback ratio calculation at the cost of additional space usage.

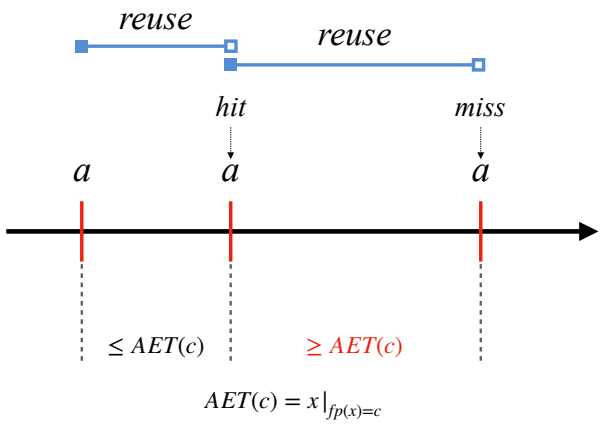

Figure 1: A visual explanation of the core ideas behind AET modeling and the relationship between footprint and miss ratio.

3 MODELING

In the following sections we describe the following four contributions:

- Analytical modeling of access pattern *writeback ratio curves* for caching systems with *infinitely* large L2 and that model's application to Zipfian programs.
- Analytical modeling of Zipfian access pattern *writeback ratio curves* for caching systems with finitely *or* infinitely large L2 and that model's application to Zipfian programs.

Both writeback models are fully general, functioning for programs with arbitrary access frequency distributions: see Propositions 1 and 2. We also instantiate each model for Zipfian frequency distributions, providing closed-form results for power-law programs in the form of Theorems 1 and 2. Pseudocode implementations of these models are available in Algorithms 1 and 2. Section 3.5.5 discusses these models' runtimes, implementations, and limitations. At a high level, the disadvantage of the second model (whose domain includes the first model's domain and as such is more general) is that it is harder to work with mathematically and as such our result for Zipfian programs is less optimized. These differences are summarized in Figure 2:

All models derive the desired information directly from access pattern information and the mathematical objects such as distributions and functions that characterize it, rather than requiring trace measurement as in many prior locality analysis techniques. An alternative approach to computing the information we need would be to generate traces according to the information passed by the user (m, α, p_w: data size, Zipfian parameter, write probability) and do trace measurement. Fast trace sampling models like AET make trace sampling efficient, but the asymptotic cost of trace generation is $O(n)$ for both time and space for trace length n. A week-long trace from a single cluster at Twitter can be over 100

	Infinite L2	Finite L2	Instantiated for Zipfian programs	Optimized Zipfian instantiation
Infinite L2 writeback model **(Proposition 1)**	✓	✗	✓	✓
General writeback model **(Proposition 2)**	✓	✓	✓	✗

Figure 2: The tradeoffs between our two models. The first is less general but easier to work with mathematically (and hence optimize), while the second is more general but more complex.

GB [37]. In addition, very large traces would have to be generated to guarantee characteristics conformant with the workload probability distribution. The models are intuitively faster than trace generation or collection, and they are asymptotically faster as well: by binary searching over window sizes, two of the models in the following sections run with $O(log(\hat{n}))$ time and constant space cost (and the third with $O(m \cdot log(\hat{n}))$ time and constant space) to compute information for a specific cache size, where $\hat{n}$ is the maximum window size required to get the correct outputs of equations in the following sections. Almost always, $\hat{n} < n$, trace length. We present more runtime information in Table 1 at the end of the section, and more thorough discussion after it.

3.1 Modeling writebacks for infinite L2

Modeling writes is of interest to people concerned with cache performance: memory technologies such as Intel Optane often suffer from limited write bandwidth, write endurance, and/or read/write latency asymmetry [19]. In order to generate writeback information offline, we extend the modeling ideas discussed in the previous section to compute writeback information instead of cache miss information.

In the case where L2 is infinitely large (in practical terms, when application data fits in L2), we know that the only misses in L2 are cold misses. In other words, the only writes to L2 (excluding cold misses) are writebacks from L1, or dirty elements evicted from L1. We build a model for this case by extending the miss ratio modeling ideas from the previous section. We will first introduce the model in a fully general sense and then instantiate it for programs whose frequency distributions are Zipfian.

The main idea behind this model is to compute the probability that a given miss results in eviction of a "dirty" (written to) element from L1 (and hence a writeback to L2). To do this, we first need to compute *write footprint* ($wfp(x)$, or the expected number of dirty elements per length-x trace window.

Occurrences of element i in x accesses (window length) follow a binomial distribution with x trials and success probability $p(i)$ (access rate from the program's frequency distribution):

$$B(x, p(i)) \tag{10}$$

Expectation of a random variable X following this distribution is simple:

$$X \sim B(x, p(i)) \rightarrow E[X] = x \cdot p(i) \tag{11}$$

We now approximate $wfp(x)$ with probabilistic counting, much as in Equation 2. We compute for all data the chance that, in a window of length x, at least one of the expected accesses is a write. Mathematically:

$$wfp(x) = \sum_{i=1}^{m} \left(1 - (1 - p_w)^{x \cdot p(i)}\right) \tag{12}$$

Then, we express the "dirty" part of footprint as a ratio:

$$dfp(x) = \frac{wfp(x)}{fp(x)} \tag{13}$$

Finally, we multiply the dirty footprint ratio ($dfp(x)$) by the cache miss ratio at the point where footprint equals cache size (as in Equation 3) to determine the fraction of accesses that incur writebacks.

Proposition 1 (Infinite L2 Writeback Model). *The writeback ratio for an LRU cache of size c can be approximated as the product of miss ratio with the ratio of write footprint to footprint at the point where footprint equals cache size. Mathematically:*

$$wbr(c) = mr(c) \cdot dfp(x)|_{fp(x)=c} \tag{14}$$

This probabilistically approximates the number of misses that incur writebacks to L2. In the form presented above, the model has runtime $O(n \cdot m)$ for trace length n and data size m: we go from n to $log(n)$ by binary searching over window sizes to find the point where footprint equals cache size, and eliminate the factor of m by approximating the summation over all data elements by integrating the corresponding continuous function. Next we demonstrate this by applying the above model to Zipfian frequency distributions.

3.2 Application to Zipfian workloads

Equation 1 gives us $p(i)$ for an element with rank i. So, we can get a closed form instantiation of the previous model as follows.

Our binomial distribution and corresponding expectation look as follows:

$$B\left(x, \frac{\omega(m, \alpha)}{i^{\alpha}}\right) \tag{15}$$

Write footprint:

$$wfp(x) = \sum_{i=1}^{m} \left(1 - (1 - p_w)^{x \cdot \frac{\omega(m, \alpha)}{i^{\alpha}}}\right) \tag{16}$$

Our dirty footprint ratio:

$$dfp(x) = \frac{\sum_{i=1}^{m} 1 - (1-p_w)^{x \cdot \frac{\omega(m,\alpha)}{i^\alpha}}}{\sum_{i=1}^{m} 1 - \left(\frac{i^\alpha - \omega(m,\alpha)}{i^\alpha}\right)^x} \tag{17}$$

The denominator can be approximated without the summation by Equation 5 (the Jelenković-Berthet approximation), and we approximate the numerator (Equation 16) as follows:

$$wfp(x) \approx \int_1^m \left(1 - (1-p_w)^{x \cdot \frac{\omega(m,\alpha)}{i^\alpha}}\right) di \tag{18}$$

$$= (m-1) - (WFP(m) - WFP(1)) \tag{19}$$

where

$$WFP(i) = \int (1-p_w)^{x \cdot \frac{\omega(m,\alpha)}{i^\alpha}} di \tag{20}$$

$$= \frac{i}{\alpha} \left(-x \cdot \omega(m,\alpha) \cdot i^{-\alpha} \cdot \ln(1-p_w)\right)^{\frac{1}{\alpha}} \cdot \Gamma\left(\frac{-1}{\alpha}, -x \cdot \omega(m,\alpha) \cdot i^{-\alpha} \cdot \ln(1-p_w)\right)$$

Here, $\Gamma(s,x)$ is the incomplete gamma function:

$$\Gamma(s,x) = \int_x^\infty t^{s-1} e^{-t} dt \tag{21}$$

related to the generalized exponential integral (Equation 7) as follows:

$$E_k(x) = x^{k-1} \cdot \Gamma(1-k, x) \tag{22}$$

Equations 18 and 20 are combined to form the following theorem.

Theorem 1. *Let m, α be the datasize and parameter characterizing a Zipfian distribution. Then the expected number of elements per length-x trace window incurring a writeback on eviction for an LRU cache, assuming static write probability p_w, can be approximated by the following:*

$$wfp(x) \approx (m-1) - \frac{m}{\alpha}\left(-x \cdot \omega \cdot m^{-\alpha} \cdot \phi\right)^{\frac{1}{\alpha}} \cdot \Gamma\left(\frac{-1}{\alpha}, -x \cdot \omega \cdot m^{-\alpha} \cdot \phi\right) + \frac{1}{\alpha}\left(-x \cdot \omega \cdot \phi\right)^{\frac{1}{\alpha}} \cdot \Gamma\left(\frac{-1}{\alpha}, -x \cdot \omega \cdot \phi\right)$$

where $\phi = \ln(1-p_w)$ and $\omega = \omega(m,\alpha)$ as in Equation 1.

The previous theorem is then combined with Equations 5, 17, and 14 to generate a writeback ratio curve for specific cache sizes. Algorithm 1 demonstrates this, and includes a binary-search based optimization to make the process of finding points where $fp(x)$ equals cache size (see Proposition 1) logarithmic in trace length. Note that in the algorithm we compute miss ratio as the finite difference of footprint (i.e. $fp(x) - fp(x-1)$) for concision, while in our implementation we computed the actual derivative of the function.

Algorithm 1: Computing writeback ratio

Input : m: data size
Input : α: Zipfian exponent
Input : c: L1 cache size
Input : p_w: write probability
Output: writeback ratio for cache size c, Zipfian distribution (m, α)

```
/* Start search with window size 1                                  */
window ← 1;
search_up(window);

Function search_up(x):
fp ← compute_footprint(x, m, α);
if fp > c then
    return search_down(x/2, x);
else
    return search_up(2x);
end function

Function search_down(x, y):
fp ← compute_footprint(⌊(x+y)/2⌋, m, α);
fp' ← compute_footprint(⌊(x+y)/2⌋ − 1, m, α);
/* Check if we've found the window size we need                     */
if fp > c and fp' < c then
    /* if so, proceed                                               */
    return compute_wbr(x, fp − fp', m, α, p_w);
/* otherwise, continue searching                                    */
else if fp > c then
    return search_down(x, ⌊(x+y)/2⌋);
else
    return search_down(⌊(x+y)/2⌋, y);
end function

Function compute_normalizer(m, α):
/* Equation 6                                                       */
ω ← 1 / Σ_{i=1}^{m} 1/i^α;
return ω;

Function compute_wbr(x, mr, m, α, p_w):
/* Equation 17                                                      */
fp ← compute_footprint(x, m, α);
wfp ← compute_wfp(x, m, α, p_w);
/* Proposition 1                                                    */
return mr · wfp/fp;

Function compute_footprint(x, m, α):
/* Equation 5 (the Jelenković-Berthet approximation)                */
return m(1 − (1/α) · E_{1+1/α}(x / (H_{m,α} · m^α)));

Function compute_wfp(x, m, α, p_w):
/* Theorem 1                                                        */
return
  (m−1) − (m/α)(−x·ω·m^{−α}·φ))^{1/α} · Γ(−1/α, −x·ω·m^{−α}·φ) +
  (1/α)(−x·ω·φ)^{1/α} · Γ(−1/α, −x·ω·φ);
```

3.3 Generalized writeback modeling

In the previous model, we built on existing miss ratio models to count writes to L2 as a fraction of misses (evictions). In this section we introduce a second model which takes a different approach by counting occurrences of the scenario that creates a writeback *at write time* as opposed to *at eviction time*. The difference is understanding when an eviction causes a writeback versus understanding when a program write will eventually need to be written back. The previous model only functions for cases where data size fits in L2; in contrast, this second approach takes into consideration the fact that data accesses can cause (non cold-start) misses in L2. As such, this model is fully general.

We compute writebacks by probabilistically counting occurences of the following scenario:

A write occurs where:

(1) The *backward reuse interval* of the item being written is small enough that the item resides in L2 cache,
(2) The *write reuse interval* of the item being written is large enough that the next time the item is written, it will not be in L1 cache

Backward reuse interval (BRI) denotes the number of trace elements between the current item and its most recent past use. If the access' BRI is too large, the write causes an L2 miss, and in that case the versions in L1 and L2 agree on value and the write in question will never incur a writeback.

Write reuse interval [10] denotes the maximum reuse interval between two consecutive writes to the same item (including all reuse intervals generated by reads in between the two writes). If the access' WRI is too small, the datum will not be evicted from L1 before it is written to again (as each reuse interval is too small to force eviction), and as such the write is "absorbed" by L1 and its value will never need to be written back. This is explained visually in Figure 3.

Our first condition, mathematically:

$$P(BRI \leq AET(L2)) \tag{23}$$

where $AET(L2) = x|_{fp(x)=c}$, or x where $fp(x) = c$, similar to Equation 3.2. This is straightforward to express probabilistically for an element of a Zipfian distribution (m, α) with rank i:

$$P(BRI \leq v) = \left(1 - (1 - p_i)^v\right) \tag{24}$$

for constant v, using the same reasoning techniques as in Equation 3.1.

Our second constraint is more difficult to model, expressible as follows:

$$P(WRI > AET(L1)) \tag{25}$$

To reiterate, write reuse interval (WRI) denotes the *maximum* reuse interval that exists between two consecutive writes to the same data item. In this section we will use a_1 and a_2 to denote the memory

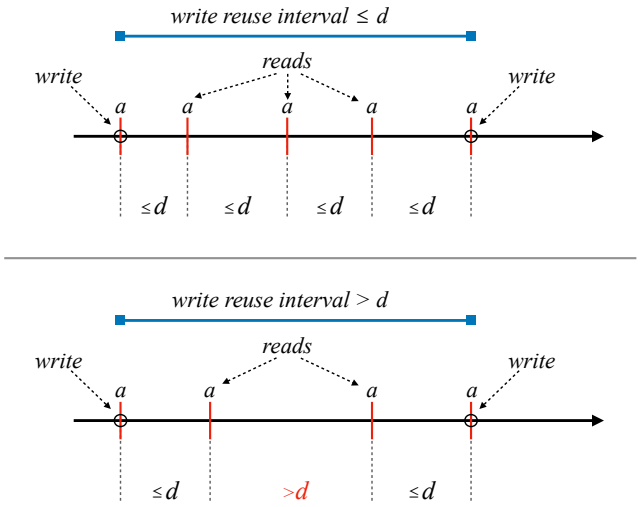

Figure 3: A visual explanation of write reuse interval, where *a* represents the data item being accessed. In the top figure, the maximum reuse interval between two consecutive writes is small enough that the item will never be evicted between them, whereas in the bottom the maximum reuse interval is too large and as such the first write needs to be written back. *d* represents a constant related to cache size, as discussed later on.

accesses forming the bounds of the write reuse pair. We observe that this can be modeled probabilistically with the following case analysis:

- **Case 1**: the next access to the item in question is a write (a_2). Here, we know there is only one reuse interval to be analyzed, as there are no reads in between the two consecutive writes forming the write reuse interval (a_1 and a_2).
- **Case 2**: the next access to the item in question is a read (a'). Here we must do further case decomposition and apply a recursive probabilistic definition.

In Case 1, which occurs with probability p_w (static write probability over the whole execution), the probability that the write reuse interval is "too big" ($WRI > d$ for $d = AET(L1)$) is equivalent to the probability that a single reuse interval is "too big" ($RI > d$):

$$P(WRI > d) = p_w \cdot \left((1 - p_i)^d\right) + (1 - p_w) \cdot P(WRI > v | \text{ Case 2}) \tag{26}$$

for constant v.

In Case 2, things are slightly more complicated. We know that the next access to the current item is a read (a'), giving us another level of case decomposition:

(1) **Case 2a**: the reuse interval formed by the next read (from a_1 to a') is "too big" ($RI > d$)
(2) **Case 2b**: the reuse interval formed by the next read is small enough ($RI \leq d$), but the write reuse interval from a' to a_2 is "too big" ($WRI > d$)

Case 2a is simple to model, just as in Equation 3.18. We observe that Case 2b can be modeled with a recursive probability definition: if we know that the reuse interval between a_1 and a' is small ($RI \leq d$), the probability that there exists a "too big" ($RI > d$) reuse interval *between* a' *and* a_2 is identical to our original probability, as the data item in question has unchanged access frequency from the Zipfian distribution. We can then rewrite Equation 3.20 as follows:

$$P(WRI > d) = p_w \cdot ((1-p_i)^d) + (1-p_w) \cdot ((1-p_i)^d + (1-(1-p_i)^d) \cdot P(WRI > d)) \tag{27}$$

where $p_i = \frac{\omega(m,\alpha)}{i^\alpha}$ as before. Expanded:

$$P(WRI > d) = P(WRI > d) \cdot (1 - p_w + (p_w - 1) \cdot ((1-p_i)^d)) + ((1-p_i)^d) \tag{28}$$

$$P(WRI > d) \cdot \left((1-p_w) \cdot \left((1-p_i)^d\right) + p_w\right) = \left((1-p_i)^d\right) \tag{29}$$

Solving for the probability we're interested in, we get the following expression:

$$P(WRI > d) = \frac{(1-p_i)^d}{(1-p_w)(1-p_i)^d + p_w} \tag{30}$$

This gives the fraction of accesses that cause a writeback of element i: as before, summing over data size will give total writeback ratio. We can then combine the probabilities from our case analyses:

Proposition 2 (General Writeback Model). *The writeback ratio of a program with write probability p_w on an LRU cache can be approximated as follows:*

$$wbr(c) = \sum_{i=1}^{m} \frac{p_w \cdot (1-(1-p_i)^v) \cdot (1-p_i)^d}{(1-p_w)(1-p_i)^d + p_w} \tag{31}$$

where $d = AET(L1)$ and $v = AET(L2)$ (see Equation 23).

3.4 Application to Zipfian workloads

Substituting Zipfian access probability into the previous equations is straightforward. The first condition:

$$P(BRI \leq v) = \left(1-(1-p_i)^v\right) = \left(1 - \left(1 - \frac{\omega(m,\alpha)}{i^\alpha}\right)^v\right) \tag{32}$$

And the second:

$$P(WRI > d) = \frac{\left(1 - \frac{\omega(m,\alpha)}{i^\alpha}\right)^d}{(1-p_w)\left(1 - \frac{\omega(m,\alpha)}{i^\alpha}\right)^d + p_w} \tag{33}$$

Combined:

Theorem 2. *Let m, α be the datasize and parameter characterizing a Zipfian distribution. Then the fraction of data accesses causing a cache writeback from L1 to a finite L2, assuming LRU caching with static write probability p_w, can be approximated by the following:*

$$wbr(d,v) = \sum_{i=1}^{m} \frac{p_w \cdot \left(\frac{\omega}{i^\alpha}\right) \cdot \left(1 - \left(1 - \frac{\omega}{i^\alpha}\right)^v\right) \cdot \left(1 - \frac{\omega}{i^\alpha}\right)^d}{(1-p_w)\left(1 - \frac{\omega}{i^\alpha}\right)^d + p_w}$$

where $\omega = \omega(m,\alpha)$, $d = AET(L1)$ and $v = AET(L2)$ (see Equation 23).

Algorithm 2 contains a pseudocode specification that applies Theorem 2 to probabilistically approximate writeback ratio. Note that, in contrast to Algorithm 1, $AET(L1)$ (and $AET(L2)$, although this of course is not meaningful for infinite L2) are inputs to the algorithm and not computed by binary search. We elide this for readability of the algorithm; add the binary search step present in Algorithm 1 to compute these.

Algorithm 2: Computing writeback ratio for finite L2

Input : m: data size
Input : α: Zipfian exponent
Input : d: AET(L1)
Input : v: AET(L2)
Input : w: write probability

ratioSum ← 0;
for $i \leftarrow 1$ *to* m **do**
  /* Theorem 2 */
  $p_i \leftarrow$ get_frequency(i);
  $BRI_i \leftarrow$ get_BRI_probability(p_i);
  $WRI_i \leftarrow$ get_WRI_probability(p_i);
  ratioSum ← ratioSum + $p_i \cdot w \cdot BRI_i \cdot WRI_i$;
return ratioSum;

Function get_frequency(i):
/* Equation 1 */
return $\frac{\omega(\alpha,m)}{i^\alpha}$;

Function get_BRI_probability(i):
/* Equation 32 */
return $\left(1 - \left(1 - \frac{\omega(m,\alpha)}{i^\alpha}\right)^v\right)$;

Function get_WRI_probability(i):
/* Equation 33 */
return $\frac{\left(1-\frac{\omega(m,\alpha)}{i^\alpha}\right)^d}{(1-w)\left(1-\frac{\omega(m,\alpha)}{i^\alpha}\right)^d + w}$;

3.5 Model Runtimes and Limitations

3.5.1 Miss ratio. We implement Berthet's miss ratio model [5] (Equation 5, the Jelenković-Berthet approximation) in Rust, using the RGSL math library [2] for the special mathematical functions required. Algorithm 1, which computes writeback ratio, contains specification for computing miss ratio, as does the original paper,

and as such we present no standalone algorithm for the miss ratio model.

3.5.2 Writeback ratio for infinite L2. Algorithm 1 contains a pseudocode specification for how this model, culminating in Theorem 1, can be implemented. Our implementation is in Rust using the RGSL math library [2]. In the Evaluation section we compare this model to a trace-based cache simulator and the AET online sampling tool. As the optimizations we present in this section require binary searching and evaluation of special mathematical functions, Algorithm 1 is quite lengthy and technical, but should provide insight on how the equations in the modeling section are combined and applied.

Theorem 1 uses $\Gamma(s, x)$ and not $E_k(x)$ (see Equation 22) because **rgsl::exponential_integrals::En()**, the GSL implementation of Equation 7, accepts only integer parameters for the subscript k which in our case is $1 + \frac{1}{\alpha}$. We convert with Equation 22 and use **rgsl::gamma_beta::incomplete_gamma::gamma_inc()** instead.

3.5.3 Writeback ratio for finite L2. Algorithm 2 contains a pseudocode specification for how this model, culminating in Theorem 1, can be implemented. Because this model is not optimized to remove the summation as the first writeback model is, it requires evaluation of no special mathematical functions and does not require RGSL. We have implemented Algorithm 2 in Rust similarly to Algorithm 1 without the use of the RGSL library.

3.5.4 Summary. Table 1 gives asymptotic runtimes for the exact models (Equations 4, 16, and Theorem 2) and the approximate models (Equations 5, 18, 20), as well as the trace generation approach discussed earlier in Section 3. All three models require evaluation of functions "at the point where" some relationship holds, and each model can find this point by binary search, leading to a logarithmic factor in runtime. Algorithm 1 contains a pseudocode implementation of this binary-search based optimization.

Our RGSL function calls contain a *do...while* loop and as such their runtime is somewhat difficult to analyze, but we consider their most expensive execution to be a constant upper bound that can be removed asymptotically when considering the time complexity of the closed form model in the following table.

These analytical models derive results directly from distributions, which is mathematically equivalent to processing infinitely long traces. In the previous table, $\hat{n}$ is the length of the shortest trace that perfectly conforms to our distributions required to derive the performance of a given cache capacity (i.e x such that $fp(x)$ is big enough to apply the Denning-HOTL conversion [[39], Equation 3]). The larger the cache, the larger $\hat{n}$ needs to be. This differs from trace analysis, for which n is the actual trace length. In all but the most contrived examples/unlikely traces, $\hat{n} < n$. We leave it to future work to more rigorously analyze this relationship and develop better understanding of $\hat{n}$.

3.5.5 Limitations. The models discussed in the previous sections, while accurate and effective, have the following limitations:

Workloads

Approach		Complexity
Model: summation	Miss ratio Writeback ratio (inf. L2) Writeback ratio (general)	$O(m \cdot s \cdot log(\hat{n}))$
Model: optimized	Miss ratio Writeback ratio (inf. L2) Writeback ratio (general)	$O(m + s \cdot log(\hat{n}))$ $O(m + s \cdot log(\hat{n}))$ N/A
Trace Analysis (AET)	Miss ratio Writeback ratio (inf. L2) Writeback ratio (general)	$O(n)$
Trace Analysis (simulation)	Miss ratio Writeback ratio (inf. L2) Writeback ratio (general)	$O(s \cdot n)$

Table 1: Asymptotic complexity of different approaches for computing the miss and writeback information that CLUsivity requires. n, m are tracelength and datasize. $\hat{n}$ is described in the following paragraph. s denotes the number of cache sizes for which writeback information needs to be generated.

These models derive miss and writeback information from frequency distributions, with the caveat that the underlying distribution must be Zipfian. The methods by which the exact, intractable expressions (Equations 4, 16, 3.25) are derived are fully general, but the applicable versions are specific to Zipfian distributions. Trace analysis is more general than these models and applicable to all workloads. Chen et al. [10] developed a model of write locality based on reuse distance. Reuse distance can be analyzed in near-linear time [41] and in logarithmic space [34].

Static write probability

The models assume static write probability p_w across all data elements: in other words, they assume that the percentage of memory accesses that are writes for the whole execution is the same as the per-element percentage. This of course can be untrue in real applications, as write hotness can vary across an application's data. We leave it to future work to consider variance in write probability.

Caching systems

The caching systems here are assumed to be using the LRU replacement policy. While for many caching systems (especially hardware caching) this will lead to inaccuracies in prediction results, LRU is commonly used in software caching systems (e.g. memcached [1]) and is perhaps the most representative single policy across all caching systems. Attempts to extend the modeling ideas here to other policies, while appealing, could be difficult due to the mathematical bases for the models [5, 39] both considering LRU.

The two writeback models proposed considered both apply only to two level, vertically aligned caching systems. We leave consideration of more complicated cache topologies to future work.

Trace Parameters			
Zipf-α	% Writes	Objects	Number of requests
0.8, 1.0, 1.2	0.05, 0.25, 0.50	10M	500M

Table 2: Trace parameters used in evaluation. Zipfian alpha and writes are chosen to reflect a realistic range from typical key-value workloads.

4 EVALUATION

4.1 Experimental Setup

Our experiments are run on a Dell PowerEdge R730 server with 8-core Intel Xeon CPU (E5-2620) and 128GB DRAM running Ubuntu 16.04 with Linux kernel 4.4.0. The AET tool is written in C compiled with gcc 5.4 (-O3). The trace generator is written in Python using Numpy libraries. The simulator is based on libCacheSim [36] written in C++ compiled with g++ 5.4 (-O3). The interested reader can find our source code here: `github.com/anon/model-tools`.

Evaluation Traces. To evaluate each model, we compared model-generated writeback ratios with ground truth writeback ratios and those generated by the AET online sampling tool on nine traces each, varying Zipfian α, write probability p_w, and L2 size for the finite L2 case. These traces are synthetic traces, generated to be Zipfian with uniform write probability. Table 2 details the parameters used in the traces. We assume uniform object size across all traces since miss ratio curves are often used to describe a class of objects relating to a particular size bound [8, 16, 31].

We choose to use synthetic traces because, given our stated limitations/assumptions in Section 3.5.5, the best way to evaluate the accuracy of our models without external sources of error is to apply them to cases where our assumptions are met. Analyzing the performance of our models on real-world industry cloud caching traces (e.g. Twitter cache traces [37]) would not be an effective form of evaluation, as it would be difficult to decouple the error inherent in the model due to our probabilistic and trace agnostic approach from the error due to the trace's deviance from its mathematical characterization (accuracy of the Zipfian α parameter, write probability variance across keys, validity of the IRM assumption/churn, etc). We recognize the utility of being able to apply these models to traces that don't fully conform to our assumptions, and leave a rigorous analysis of our models' performance on industry traces and how to correct for external sources of error to future work.

4.2 Modeling writebacks for infinite L2

Our first writeback model is extremely accurate: Table 3 presents numerical percent deviation from ground truth results for our predicted writeback ratios as well as those generated by the AET online trace sampling technique. We see that, even without requiring a trace, our model slightly outperforms the accuracy of AET, with average errors of 5.0 and 5.2% respectively. For higher α and P(write) values the model error tends to be higher: this is a result of more variance reducing the effectiveness of our assumptions about the trace.

α	**0.8**		**1**		**1.2**	
P(write)	Anl.	AET	Anl.	AET	Anl.	AET
0.05	0.1	0.6	1.4	1.0	2.9	2.3
0.25	1.8	2.6	3.2	4.1	5.2	6.7
0.5	10.6	7.4	10.2	9.1	9.6	13.2

Table 3: WBR modeling for infinite L2. Relative differences (percent deviation from ground truth) for the model's writeback ratio prediction compared to the AET sampling technique across different static write probabilities and αs. The baseline is an offline cache simulator. Lower is better.

4.3 Generalized writeback modeling

Our second model is very accurate as well, this time with an average error of 3.0% compared to AET's 2.7%. Note that AET performs better here than in the previous model: this is a result of our building additional functionality into the AET model to allow for an apples-to-apples comparison (see the AET background section). Table 4 presents numerical results for the generalized writeback model.

α	**0.8**		**1**		**1.2**	
P(write)	Anl.	AET	Anl.	AET	Anl.	AET
0.05	0.2	1.8	0.0	2.5	4.2	1.2
0.25	0.3	0.9	1.3	1.1	8.4	2.6
0.5	0.5	0.7	1.9	1.1	10.1	1.6

Table 4: Generalized WBR modeling. Relative differences (percent deviation from ground truth) for the model's writeback ratio prediction compared to the AET sampling technique across different static write probabilities and αs. The baseline is an offline cache simulator. Lower is better.

4.4 Runtimes and speedup

Table 5 presents the execution times for trace generation, AET, our analytical models, and the simulator. Note that AET and the simulator require a trace, so we report both the runtime of AET for cases where the user has a pre-existing trace and the cost of generating a trace. Note that AET generates full miss and writeback ratio curves sampled from the trace, while the simulator must be run for each individual L1 and L2 size. We simulate 16 cache sizes for each trace and report an estimated 1137 minute (19 hour) execution time for 100 cache sizes (s=100). Likewise, the analytical models are run for individual L1 and L2 sizes in order to generate sufficient miss ratio and writeback ratio information. The generalized model's asymptotic runtime is $O(m \cdot s \cdot log(\hat{n}))$, where s is the number of cache sizes for which writeback information is computed. This is reflected in the 200x runtime increase from $s = 1$ to $s = 100$ (note that a writeback ratio is defined for each L1/L2 cache size combination). The infinite L2 model, on the other hand, runs in $O(m + s \cdot log(\hat{n}))$, where it is typical that $m >> s \cdot log(\hat{n})$, so similar executions time for one cache size (s=1) and many cache sizes (s=100) are expected for this model.

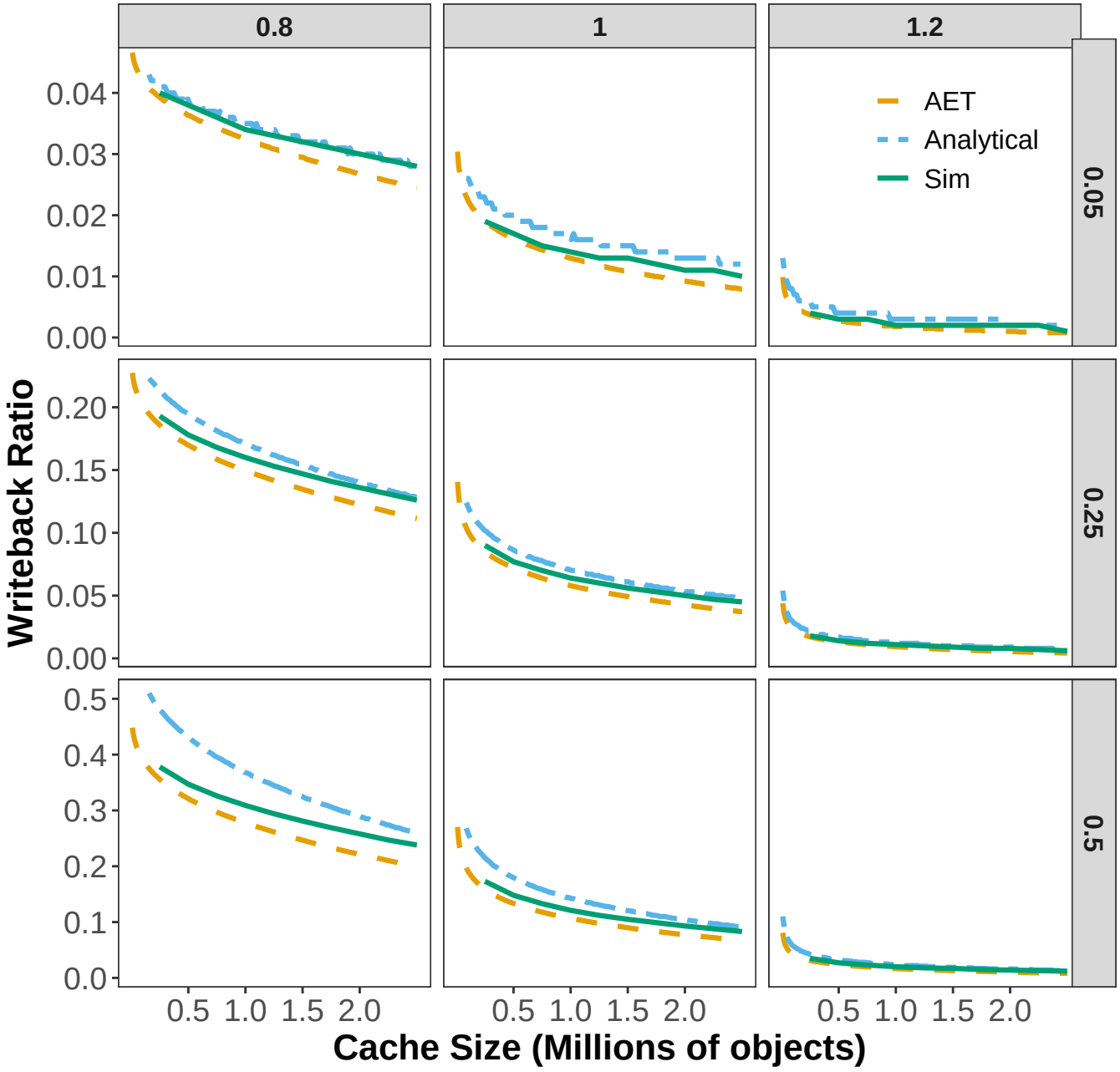

Figure 4: WBR modeling for infinite L2. Writeback ratio comparison for our model (Analytical) against an online trace analysis model (AET) and an offline cache simulator (Sim) which represents ground truth. The values at the top of each column represent varied α values, while the values at the right side of each row represent varied write probability values.

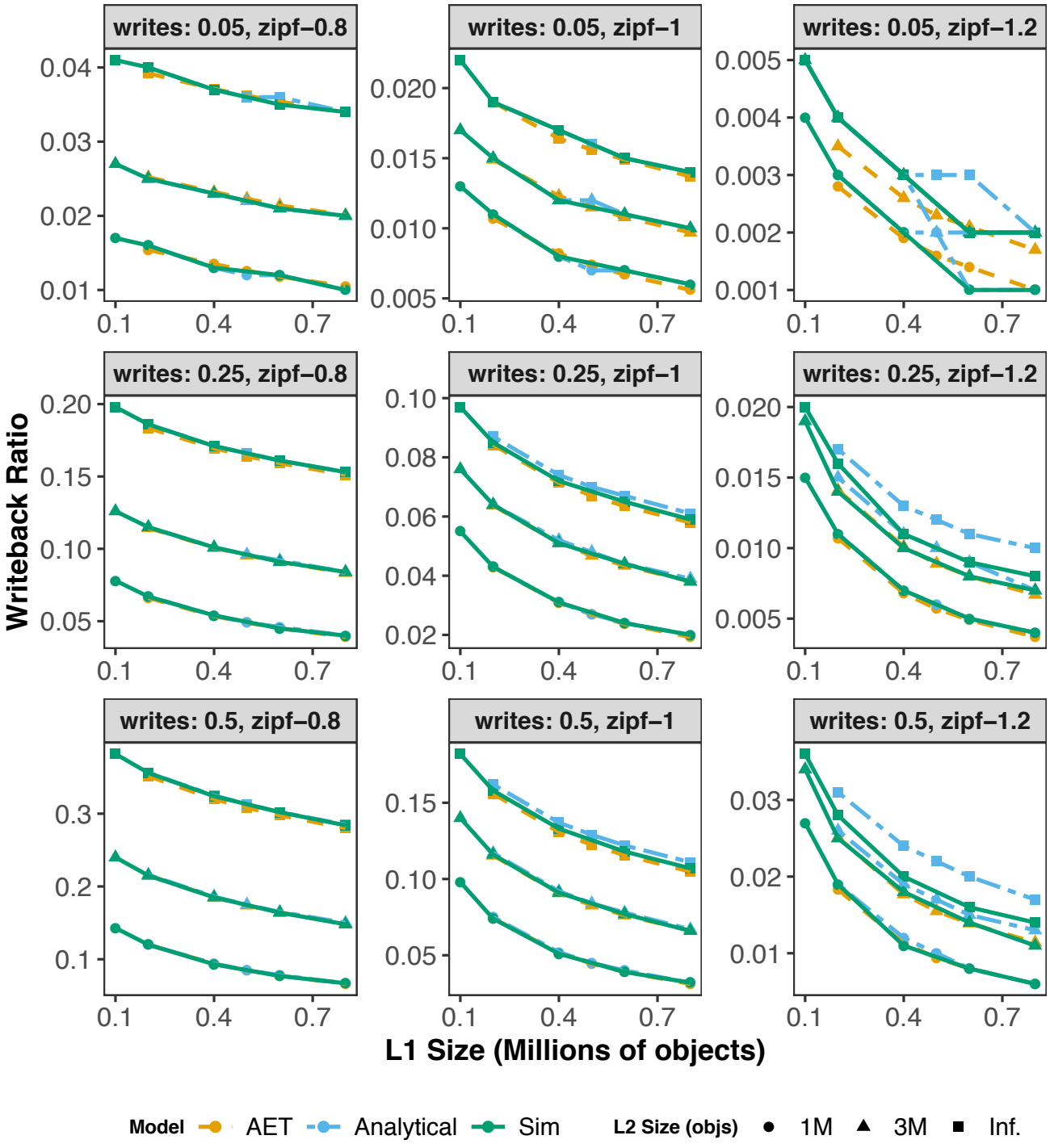

Figure 5: Generalized writeback modeling Writeback ratio comparison for our model (Analytical) against an online trace analysis model (AET) and an offline cache simulator (Sim), which represents ground truth. The values of α and write probability appear at the top of each graph. The three sets of lines per graph represent varied L2 size, the largest of which is infinite (data size fits in L2).

Method	Requires Trace	# of Cache Sizes	
		s = 100	s = 1
AET	yes	4m 33s	4m 33s
Simulator	yes	1137m (est.)	36m 24s
Generalized Anl. Model	no	5m 46s	2.6s
Infinite L2 Anl. Model	no	0.54s	0.53s
Trace Generation Time		46m 51s	

Table 5: Execution times for trace generation and each method of obtaining writeback ratio information. Average execution times are over the 9 synthetic traces used in the evaluation.

Model	Trace pre-existing (AET)	Speedup over AET
Infinite L2 Anl.	yes	515x
	no	5,819x
Generalized Anl.	yes	105x
	no	1,186x

Table 6: Speedup information: single cache size calculation ($s = 1$)

Model	Trace pre-existing (AET)	Speedup over AET
Infinite L2 Anl.	yes	506x
	no	5,711x
Generalized Anl.	yes	0.8x
	no	8.9x

Table 7: Speedup information: miss ratio curve calculation ($s = 100$)

Tables 6 and 7 contain the previous runtime information in speedup form. Specifically, they compare the two analytical models to AET (as that is a faster and more realistic method than simulation to obtain writeback information) both when AET does and does not have a preexisting trace to analyze.

5 RELATED WORK

5.1 Locality models

Yuan et al. [39] introduce a unifying mathematical framework that rigorously defines and analyzes different approaches to measuring program locality. Their highest-level decomposition separates locality analyses into the following three categories: *access locality*, related to individual memory accesses; *timescale locality*, related to understanding sequences of memory accesses; and *cache locality*, related to cache behavior. This work is primarily concerned with the interactions between program frequency distributions and caching systems, falling under the access locality and cache locality categories. Most methods of frequency analysis are built on top of the Independent Reference Model (IRM) [32], a simplistic but effective model in which memory accesses are independent random variables sampled from a given distribution. Another technique in the access locality category is Static Parallel Sampling [9]. Like our work, SPS generates predictive statistics for an program's execution, although SPS generates reuse information through compile-time analysis of program references while we generate cache statistics through a fully offline analysis of the program's workload characterization information.

5.2 Write Locality

The research community interested in caching systems recognizes the importance of considering the differences between reads and writes. For example, Beckmann et al. [4] describe how to integrate writeback cost into cache replacement algorithms. The main reason for this consideration is hardware limitations: memory technologies such as Intel Optane often suffer from limited write bandwidth, write endurance, and/or asymmetric read/write latency and power consumption [19]. A related issue is *write amplification* [33], where storage systems write substantially more data than necessary as a result of data granularity and wear leveling. Yang et al. [38] build an analytical model of write amplification in block I/O traces that is very similar in motivation and goal to the models presented in this work, although different in technique and domain. Salkhordeh et al. [28] analytically model the lifetime and performance of Intel Optane modules, taking into consideration their fundamental asymmetries for reads and writes and their wear leveling technique [15], which Intel argues gives Optane SSDs several times the endurance of NAND SSDs. Olson and Hill [24] design a cache replacement policy using mathematical approximations of working-set size to reduce writeback count.

Chen et al. [10] introduce a metric for measuring write locality, *write reuse interval*, that is central to one of our writeback models. Write reuse interval is defined as follows: let a_1, a_2 be two consecutive writes to the same data item (a) in a trace. Between a_1 and a_2 there exist any number of accesses to other data items, as well as any number of *reads* to a (but no writes to a). If *reuse interval* [39] denotes the number of accesses between an access to an item and the next access to the same item, what is the maximum reuse interval formed by a_1, a_2, and any of the reads to a in between the two?

The same paper introduces a linear-time algorithm for computing writeback ratio based on trace analysis. Our analytical models provide asymptotically faster methods of computing the same information without the need for traces.

Kant [22] introduces an analytical model capable of approximating writeback counts in multiprocessor systems: however, this model uses iterative algorithms on Markov chain formulations and as such does not introduce closed-form solutions as we do. Their focus is on approximating writebacks in the context of the MESI coherence protocol on hardware, while we focus on the problem of approximating writebacks from frequency distributions, especially Zipfian distributions. As a result the domain and results of their work are fundamentally different from ours.

5.3 Miss ratio modeling

In the past few years, the Content-Centric Network model has gained steam in the networking community, and as a result much work has been done to analytically model LRU miss ratios and throughputs for various workload types, interleavings, and cache topologies [7, 14, 21, 27]. These works, while similar in approach and technique, do not consider writes and writebacks, which are of growing performance importance due to memory hardware limitations and are the main focus of our paper.

5.4 Analysis of Zipfian distributions

Mathematical analysis of Zipfian distributions is standard in research in related areas. Serpanos et al. [29] derive a probabilistic bound on how much of an execution needs to be observed to determine a workload's unknown α value. In the context of cache design, Breslau et al. [6] derive reuse behavior from Zipfian distributions and argue for specific approaches to frequency-based caching, and Li et al. [23] develop a device-to-device (D2D) caching scheme for Zipfian distributions based on an analytical expression of edge cache hit rate.

The base Zipfian model has been extended for purposes beyond those listed above: for example, Duarte-Lopez et al. [11] introduce what they term the Zipf-Poisson-stopped-sum distribution, whose added complexity over the standard power-law distribution makes it more representative of real-world social network graphs.

6 CONCLUSION

The prevalence of data-bound applications means that understanding memory performance is more important than ever. More specifically, the rise of write-bottlenecked memory technologies such as Optane means we must go beyond traditional caching analyses that focus on hit rate. We have introduced two novel models that predict cache writebacks in LRU caching systems. The first functions when application data size fits in L2 cache and counts occurences of eviction-time situations that trigger writebacks. The second is more general, working for infinite or finite L2, and counts occurences of access-time situations that will eventually incur a writeback. We instantiate both models with frequency distributions from Zipfian access patterns to derive two fully analytical models, capable of predicting writeback ratio with no tracing or sampling required. We demonstrate that these models are extremely accurate and fast: the first model, for infinitely large L2 systems, averaged 5.0% relative error from ground truth and achieved a minimum speedup over the AET trace analysis technique of 515x to generate writeback information for a single cache size. The second model, which is fully general with respect to L1 and L2 sizes but slower, averaged 3.0% relative error from ground truth and achieved a minimum speedup over AET of 105x for a single cache size. Taken together, the two writeback models and their applications to Zipfian programs represent both a theoretical contribution to our ability to understand memory performance and a practical contribution to our ability to design effective caching systems for domains including graph analytics and key-value stores.

ACKNOWLEDGMENTS

Thanks to the members of the systems group at the University of Rochester who gave valuable feedback on a version of this work.

The research is supported in part by the National Science Foundation (Contract No. CCF-2114319, CNS-1909099, CCF-1717877). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding organizations.

REFERENCES

[1] 2018. memcached. https://memcached.org.
[2] 2020. RGSL Library. https://docs.rs/GSL/1.1.0/rgsl/index.html.
[3] Lada A Adamic and Bernardo A Huberman. 2002. Zipf's law and the Internet. *Glottometrics* 3, 1 (2002), 143–150.
[4] Nathan Beckmann, Phillip B Gibbons, Bernhard Haeupler, and Charles McGuffey. 2020. Writeback-aware caching. In *Symposium on Algorithmic Principles of Computer Systems (APOCS 2020)*.
[5] Christian Berthet. 2017. Approximation of LRU Caches Miss Rate: Application to Power-law Popularities. *CoRR* abs/1705.10738 (2017). arXiv:1705.10738 http://arxiv.org/abs/1705.10738
[6] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. 1998. *On the implications of Zipf's law for web caching*. Technical Report. University of Wisconsin-Madison Department of Computer Sciences.
[7] Giovanna Carofiglio, Massimo Gallo, Luca Muscariello, and Diego Perino. 2011. Modeling data transfer in content-centric networking. In *2011 23rd International Teletraffic Congress (ITC)*. 111–118.
[8] D. Carra and P. Michiardi. 2014. Memory partitioning in Memcached: An experimental performance analysis. In *2014 IEEE International Conference on Communications (ICC '14)*. 1154–1159.
[9] Dong Chen, Fangzhou Liu, Chen Ding, and Sreepathi Pai. 2018. Locality Analysis through Static Parallel Sampling. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 557–570. https://doi.org/10.1145/3192366.3192402
[10] Dong Chen, Chencheng Ye, and Chen Ding. 2016. Write Locality and Optimization for Persistent Memory. In *Proceedings of the Second International Symposium on Memory Systems* (Alexandria, VA, USA) *(MEMSYS '16)*. Association for Computing Machinery, New York, NY, USA, 77–87. https://doi.org/10.1145/2989081.2989119
[11] Ariel Duarte-López, Marta Pérez-Casany, and Jordi Valero. 2020. The Zipf–Poisson-stopped-sum distribution with an application for modeling the degree sequence of social networks. *Computational Statistics & Data Analysis* 143 (2020), 106838. https://doi.org/10.1016/j.csda.2019.106838
[12] Ronald Fagin. 1977. Asymptotic Miss Ratios over Independent References. *J. Comput. Syst. Sci.* 14, 2 (1977), 222–250. https://doi.org/10.1016/S0022-0000(77)80014-7
[13] Dror Feitelson. 2014. *Workload Modeling for Computer Systems Performance Evaluation*. Cambridge University Press.
[14] Christine Fricker, Philippe Robert, James Roberts, and Nada Sbihi. 2012. Impact of traffic mix on caching performance in a content-centric network. In *2012 Proceedings IEEE INFOCOM Workshops*. 310–315. https://doi.org/10.1109/INFCOMW.2012.6193511
[15] Frank Hady. 2020. *Intel Optane Technology Delivers New Levels of Endurance*. Technical Report. Intel Corp.
[16] 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)*. USENIX Association, Santa Clara, CA, 57–69.
[17] Xiameng Hu, Xiaolin Wang, Lan Zhou, Yingwei Luo, Chen Ding, and Zhenlin Wang. 2016. Kinetic Modeling of Data Eviction in Cache. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference* (Denver, CO, USA) *(USENIX ATC '16)*. USENIX Association, USA, 351–364.
[18] Xiameng Hu, Xiaolin Wang, Lan Zhou, Yingwei Luo, Zhenlin Wang, Chen Ding, and Chencheng Ye. 2018. Fast Miss Ratio Curve Modeling for Storage Cache. *ACM Trans. Storage* 14, 2, Article 12 (April 2018), 34 pages. https://doi.org/10.1145/3185751
[19] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. 2019. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. *CoRR* abs/1903.05714 (2019). arXiv:1903.05714 http://arxiv.org/abs/1903.05714
[20] Predrag R. Jelenković. 1999. Asymptotic approximation of the move-to-front search cost distribution and least-recently used caching fault probabilities. *Ann. Appl. Probab.* 9, 2 (05 1999), 430–464. https://doi.org/10.1214/aoap/1029962750

[21] Zixiao Jia, Peng Zhang, Jiwei Huang, Chuang Lin, and John C. S. Lui. 2013. Modeling Hierarchical Caches in Content-Centric Networks. In *2013 22nd International Conference on Computer Communication and Networks (ICCCN)*. 1–7. https://doi.org/10.1109/ICCCN.2013.6614153

[22] Krishna Kant. 2009. Estimation of Invalidation and Writeback Rates in Multiple Processor Systems. (04 2009).

[23] Qiang Li, Yuanmei Zhang, Ashish Pandharipande, Xiaohu Ge, and Jiliang Zhang. 2019. D2D-assisted caching on truncated Zipf distribution. *IEEE Access* 7 (2019), 13411–13421.

[24] Lena Olson and Mark Hill. 2016. Probabilistic Directed Writebacks for Exclusive Caches. 44, 1 (jul 2016). https://doi.org/10.1145/2971331.2971334

[25] Ivy Bo Peng, Maya B. Gokhale, and Eric W. Green. 2019. System evaluation of the Intel optane byte-addressable NVM. In *Proceedings of the International Symposium on Memory Systems, MEMSYS 2019, Washington, DC, USA, September 30 - October 03, 2019*. ACM, 304–315. https://doi.org/10.1145/3357526.3357568

[26] David MW Powers. 1998. Applications and explanations of Zipf's law. In *New methods in language processing and computational natural language learning*.

[27] Elisha Rosensweig and Jim Kurose. 2013. A Network Calculus for Cache Networks. *Proceedings - IEEE INFOCOM*, 85–89. https://doi.org/10.1109/INFCOM.2013.6566740

[28] R. Salkhordeh, O. Mutlu, and H. Asadi. 2019. An Analytical Model for Performance and Lifetime Estimation of Hybrid DRAM-NVM Main Memories. *IEEE Trans. Comput.* 68, 8 (2019), 1114–1130. https://doi.org/10.1109/TC.2019.2906597

[29] D. N. Serpanos, G. Karakostas, and W. H. Wolf. 2000. Effective caching of Web objects using Zipf's law. In *2000 IEEE International Conference on Multimedia and Expo. ICME2000. Proceedings. Latest Advances in the Fast Changing World of Multimedia (Cat. No.00TH8532)*, Vol. 2. 727–730 vol.2. https://doi.org/10.1109/ICME.2000.871464

[30] Daniel G. Waddington, Mark Kunitomi, Clem Dickey, Samyukta Rao, Amir Abboud, and Jantz Tran. 2019. Evaluation of intel 3D-xpoint NVDIMM technology for memory-intensive genomic workloads. In *Proceedings of the International Symposium on Memory Systems, MEMSYS 2019, Washington, DC, USA, September 30 - October 03, 2019*. ACM, 277–287. https://doi.org/10.1145/3357526.3357528

[31] Carl A. Waldspurger, Trausti Saemundsson, Irfan Ahmad, and Nohhyun Park. 2017. Cache Modeling and Optimization using Miniature Simulations. 487–498. https://www.usenix.org/conference/atc17/technical-sessions/presentation/waldspurger

[32] Wikipedia contributors. 2020. Independent Reference Model — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Independent_Reference_Model&oldid=959913121 [Online; accessed 7-April-2021].

[33] Wikipedia contributors. 2021. Write amplification — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Write_amplification&oldid=1015893909 [Online; accessed 19-April-2021].

[34] 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.

[35] Cong Xie, Ling Yan, Wu-Jun Li, and Zhihua Zhang. 2014. Distributed Power-law Graph Computing: Theoretical and Empirical Analysis.. In *Nips*, Vol. 27. 1673–1681.

[36] Juncheng Yang. 2021. *libCacheSim - a library for cache simulation, profiling, and analysis*. Retrieved May, 2021 from https://github.com/1a1a11a/libCacheSim

[37] Juncheng Yang, Yao Yue, and K. V. Rashmi. 2020. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 191–208. https://www.usenix.org/conference/osdi20/presentation/yang

[38] Yue Yang and Jianwen Zhu. 2016. Write Skew and Zipf Distribution: Evidence and Implications. *ACM Trans. Storage* 12, 4, Article 21 (June 2016), 19 pages. https://doi.org/10.1145/2908557

[39] Liang Yuan, Chen Ding, Wesley Smith, Peter Denning, and Yunquan Zhang. 2019. A Relational Theory of Locality. *ACM Trans. Archit. Code Optim.* 16, 3, Article 33 (Aug. 2019), 26 pages. https://doi.org/10.1145/3341109

[40] Teng Zhang, Jianying Wang, Xuntao Cheng, Hao Xu, Gui Huang, Tieying Zhang, Dengcheng He, Feifei Li, Wei Cao, Zhongdong Huang, and Jianling Sun. 2020. FPGA-Accelerated Compactions for LSM-based Key-Value Store. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST '20)*.

[41] Yutao Zhong, Xipeng Shen, and Chen Ding. 2009. Program Locality Analysis Using Reuse Distance. *ACM Trans. Program. Lang. Syst. (TOPLAS '09)* 31, 6, Article 20 (Aug. 2009), 39 pages.