Modeling Cache Performance Beyond LRU

Nathan Beckmann Daniel Sanchez
Massachusetts Institute of Technology
{beckmann,sanchez}@csail.mit.edu

ABSTRACT

Modern processors use high-performance cache replacement policies that outperform traditional alternatives like least-recently used (LRU). Unfortunately, current cache models do not capture these high-performance policies as most use stack distances, which are inherently tied to LRU or its variants. Accurate predictions of cache performance enable many optimizations in multicore systems. For example, cache partitioning uses these predictions to divide capacity among applications in order to maximize performance, guarantee quality of service, or achieve other system objectives. Without an accurate model for high-performance replacement policies, these optimizations are unavailable to modern processors.

We present a new probabilistic cache model designed for high-performance replacement policies. It uses absolute reuse distances instead of stack distances, and models replacement policies as abstract *ranking functions*. These innovations let us model arbitrary age-based replacement policies. Our model achieves median error of less than 1% across several high-performance policies on both synthetic and SPEC CPU2006 benchmarks. Finally, we present a case study showing how to use the model to improve shared cache performance.

1. INTRODUCTION

On-chip caches are fundamental to cope with the long latency, high energy, and limited bandwidth of main memory accesses. Most cache space is consolidated in a large last-level cache (LLC) shared among cores, which consumes significant resources (e.g., over 50% of chip area [32]). Rising core counts place mounting pressure on the memory system, and have led to a resurgence of cache architecture research. As a result, modern LLCs have become quite sophisticated in their array organization and replacement policy.

Understanding the LLC's behavior is critical to achieve system objectives. Accurate predictions of cache behavior enable a large number of optimizations, including of single-threaded performance [5, 15, 38], shared cache performance [3, 37, 41, 50], fairness [37, 40], quality of service [3, 26], security [39], etc. These optimizations can be performed in many ways, e.g. by explicitly partitioning the shared cache in hardware [2, 34, 45] or software [11], or through job scheduling to avoid interference in cache accesses [14, 35, 56].

Unfortunately, cache behavior is difficult to predict because it depends on many factors, both of the application (e.g., its access pattern) and the cache (e.g., its size, associativity, and replacement policy). Existing cache models [1, 15, 36, 47, 55] tend to focus on traditional, set-associative caches using simple replacement policies like least-recently used (LRU), pseudo-LRU, or random replacement. *But modern processors do not use LRU (or pseudo-LRU) for the LLC.*

Modern LLCs instead employ high-performance replacement policies that greatly improve cache performance over traditional policies like LRU (Sec. 2.2). These designs are available in commercial processors [22, 53], and a new model is needed to understand their behavior.

We present a cache model that accurately predicts the behavior of high-performance replacement policies on modern LLCs. Our model leverages two key observations: First, each core's private cache hierarchy filters accesses before they reach the LLC, capturing successive references to the same address [5, 28, 30]. Thus, LLC accesses are free of short-term temporal correlations. (This is also why LRU is a poor policy for the LLC; LRU relies on temporal locality, and at the LLC, there is little.) Second, modern LLCs use hashing to map lines to sets [31, 53], reducing hotspots. Existing models often focus on the effects of low associativity, i.e. conflict misses, which can be hard to capture. But with hashing, modern LLCs have near-uniform behavior across sets and high effective associativity [44], making conflict misses a second-order concern (Sec. 2.1).

These two observations mean that *modern LLCs can be modeled as a pseudo-random process*: by capturing highly correlated accesses, private caches essentially randomize the accesses seen by the LLC; and hashing and high effective associativity mean that the replacement candidates constitute a representative sample of cached lines. Our model therefore computes the probability of various events (hits and evictions) occurring to *individual lines* as they age. This approach lets us model arbitrary age-based policies, including familiar policies and several high-performance policies, and abstracts away the details of array organization. Specifically, in this paper we evaluate LRU, PDP [16], and IRGD [51].

These features set our model apart from existing models. Existing models target set-associative LRU caches, and many model the performance of *entire sets* using *stack distances*, which measure the number of unique addresses between references to the same line. Stack distances are meaningful for set-associative LRU caches, but have little meaning for other cache designs. In particular, it is unclear how to model high-performance policies through stack distances (Sec. 2.3).

Our model is built from logically distinct components, each of which captures a separate aspect of cache behavior (Sec. 4). For instance, to model a new replacement policy, our model only requires simple changes to a single parameter, the ranking function (Sec. 5). We present an efficient, practical implementation of our model (Secs. 6 and 7) and thoroughly validate it against synthetic and real benchmarks (Sec. 8). Finally, we present a case study of cache partitioning with high-performance replacement, where our model

978-1-4673-9211-2/16/$31.00 ©2016 IEEE

enables performance gains far exceeding those of cache partitioning or high-performance replacement alone (Sec. 9). In summary, we offer an efficient way to predict the performance of high-performance policies, allowing them to enjoy the many benefits that prior work has demonstrated for LRU.

2. BACKGROUND

We first review the relevant background in modern LLC architecture, replacement policies, and cache modeling.

2.1 Modern cache architecture

Modern processors feature multilevel cache hierarchies. Each core has one or two levels of small, fast, private caches (L1 and L2). These are backed by a much larger last-level cache (LLC) that contains the bulk of cache capacity.

Cache architecture varies greatly across levels. The purpose of the private levels is to cache the program's most frequently accessed data at low latency and high bandwidth, so they are kept small and simple—and therefore fast. Specifically, they adopt a familiar, set-associative architecture and use simple replacement policies like pseudo-LRU.

The cache architecture is more sophisticated at the LLC. Since off-chip misses are very expensive, various caching techniques are attractive for the LLC that are a poor design for the private levels. The most significant change is the replacement policy, discussed in detail below. In addition, the LLC adopts an array organization that ensures high effective associativity. For instance, by hashing addresses before indexing the array, the cache pseudo-randomly spreads accesses across sets and reduces conflicts [31, 32, 49, 53]. Other designs further increase effective associativity without adding ways. For instance, skew-associative caches [48] and zcaches [44] use a different hash function for each way, mixing replacement candidates across sets.

For the purposes of cache modeling, there are two important implications. First, private caches capture most "hot" data, so the LLC's access stream is stripped of short-term temporal correlations: a second access to the same line requires that the line be first evicted from the L1 and L2. Second, since modern LLCs use hashing and achieve high effective associativity, replacement candidates form a representative sample of cached lines [5, 41, 44]. Details of array organization, which have been the focus of prior studies [1, 15, 36, 47, 55], are relatively unimportant in modern LLCs. Our model leverages this insight by modeling replacement as a probabilistic process affecting individual lines, not sets.

2.2 Replacement policies beyond LRU

The largest single difference between the traditional caches targeted by existing models and modern LLCs is the replacement policy. Traditional replacement policies (e.g., LRU, pseudo-LRU, least-frequently used, or random) all perform poorly at the LLC because private caches filter out most locality [28]. LRU is the most common, but has common performance pathologies. For instance, when an application's working set does not fit in the cache, LRU causes thrashing and performance cliffs; e.g., iterating sequentially over a 1 MB array gets zero hits with less than 1 MB of cache, but every access suddenly hits at 1 MB.

Such pathologies are avoided in the optimal replacement policy, Belady's MIN [8, 36], which relies on a perfect oracle to replace the line that will be reused furthest in the future. Prior work has proposed many mechanisms that improve on LRU and attempt to emulate optimal replacement. We observe that, details aside, they use three broad techniques:

- **Recency:** Recency prioritizes recently used lines over old ones. LRU uses recency alone, leading to its pathologies (e.g., thrashing). Most high-performance policies combine recency with other techniques.
- **Protection:** When the working set does not fit in the cache, some policies protect a portion of the working set against eviction to avoid thrashing. This is equivalent to *thrash-resistance* [23, 42].
- **Classification:** Some policies divide accesses into separate classes, and treat lines of each class differently. For example, several policies classify lines as either reused or non-reused [23, 30]. Classification works well when classes have markedly different access patterns.

For instance, DIP [42] enhances LRU by dynamically detecting thrashing using set dueling, and protects lines in the cache by inserting most lines at low priority in the LRU chain. DIP inserts a fixed fraction of lines ($\varepsilon = 1/32$) at high priority to avoid stale lines. DRRIP [23] classifies between reused and non-reused lines by inserting lines at medium priority, includes recency by promoting lines on reuse, and protects against thrashing with the same mechanism as DIP. SHiP [54] extends DRRIP with more elaborate classification, based on the memory address, program counter (PC), or instruction sequence. Instruction-based reuse distance prediction [27] uses PC-based classification, but does not do protection. PDP [16] decides how long to protect lines based on the reuse distance distribution, improving on DIP, but does not do classification. SDBP [29] and PRP [13] predict the probability candidates will hit and prioritize those that are more likely. Finally, IRGD [51] adapts its policy to the access stream, ranking lines according to their harmonic expected reuse distance. The policies we consider are described in greater detail in Sec. 5.1.

Most of these policies are age-based: lines enter the cache at a given priority, which changes over time according to how long the line has been in the cache. However, the policies vary in details. PDP, PRP, and IRGD directly count the number of accesses since a line was last referenced, while DRRIP and SHiP use a clever aging mechanism that allows them to rank lines using fewer bits in hardware. Our model targets arbitrary age-based policies, and we age lines by the number of accesses since their last reference (like PDP, PRP, and IRGD).

No prior work gives a general way to predict these policies. DIP gives a simple performance analysis on a single access pattern (the scanning pattern discussed above), but does not consider performance on realistic access patterns, nor does their analysis generalize to such cases. PDP employs an analytical model to choose for how long to protect lines in the cache. PDP's model is simple, but it is inaccurate on common access patterns (Sec. 8.1). Moreover, even if it were accurate, it is limited to modeling protecting distances, and therefore does not solve the general problem of modeling high-performance cache replacement.

2.3 Cache modeling

Most prior work has developed analytical cache models using *stack distance distributions* [1, 15, 36, 47, 55]. The stack distance of an access is the number of *unique* addresses accessed after the last reference to the same address. For example, in the sequence ABCBDDA the stack distance of A is four, as B, C, and D are accessed between both accesses to A. Stack distances are meaningful for LRU: in a fully-associative LRU cache of S lines, accesses with stack distance $\leq S$ will be hits, and all others will be misses.

Stack distances are meaningful for LRU, but have little direct meaning for other policies. Our insight is that *absolute reuse distances* [47] can be used to model cache behavior for a much wider range of replacement policies. The absolute reuse distance of an access is the total number of references after the last reference to the same address. For example, in ABCBDDA the absolute reuse distance of access A is six. Absolute reuse distances correspond to lines' ages (i.e., the number of accesses since they were last referenced): a line with absolute reuse distance d will hit at age d if not evicted.

Absolute reuse distance distributions cannot be trivially translated into miss rates. *Our key innovation is to develop an analytical model that performs this translation for a broad class of policies.* Prior work has used absolute reuse distances only in a limited context: Sen and Wood [47] use absolute reuse distances to model random replacement, but use stack distances for LRU. StatCache [9] models random replacement through absolute reuse distances, and StatStack [17] builds a model for LRU, but does not consider more recent policies. By contrast, we model LRU and several recent policies through a single framework based on absolute reuse distances.

For brevity, in the rest of the paper *we refer to absolute reuse distances simply as reuse distances*. This terminology is in line with prior cache replacement papers [16, 23, 27], but note that, since prior work in analytical models focuses on stack distance, some use reuse and stack distance as synonyms [15, 47, 55].

2.4 Applications of cache models

The inability to predict high-performance policies is especially unfortunate because it precludes many optimizations, such as job scheduling to avoid interference in cache accesses [14, 35, 56] and cache partitioning.

Cache partitioning allows software to divide space among cores, threads, or types of data [3, 11, 45], enabling system-wide management of shared caches. Cache partitioning is often used to improve performance in systems with shared caches [37, 41, 50], but also has many other uses beyond performance, and is thus better thought of as an enabling technology for *software control of the cache*.

Partitioning strikes a nice balance between scratchpad memories, which yield control to software but are hard to use, and conventional hardware-only caches, which are easy to use but opaque to software. For instance, partitioning has been used to improve fairness [37, 40], implement priorities and guarantee QoS [12, 20, 26], improve NUCA designs [3, 33], and eliminate side-channel attacks [39]. Partitioning is therefore a general tool to help achieve *system-level objectives*.

To partition the cache effectively, partitioning schemes require predictions of performance at different sizes. Unfortunately, these predictions are not available for high-performance replacement policies. Talus [5] resolves this conflict by partitioning within a single access stream to yield a predictable policy with good performance. However, Talus itself requires predictions to choose the partition sizes. Our model is thus complementary to Talus, and allows partitioning schemes to partition arbitrary age-based replacement policies.

We present a case study that shows our model can effectively partition high-performance replacement policies (Sec. 9). We optimize for shared cache performance in the case study, but bear in mind this is just one example of the many applications of partitioning in current systems.

3. OVERVIEW

Fig. 1 shows the high-level design of our cache model. The input to the model is the cache architecture (its size, associativity, and replacement policy) and a concise description of the access stream. Specifically, we describe the access stream by its *reuse distance distribution*; i.e., for each distance d, how many accesses have reuse distance d.

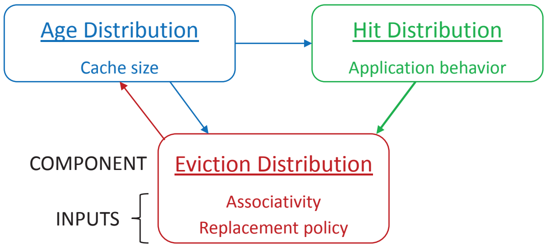


Figure 1: Our model consists of three interdependent probability distributions. Each distribution captures distinct model inputs. Arrows denote dependencies: A → B means "A depends on B".

From these inputs, our model produces a concise description of the cache's behavior. Specifically, it produces the cache's *hit and eviction distributions*: for each age a, how many accesses are hit or evicted at age a, respectively. We define *age* as the number of accesses since a line was last referenced. The hit and eviction distributions directly yield the cache's performance, as the cache's hit rate is the sum over the hit distribution. Additionally, they give a rich picture of the cache's behavior that can be used to improve its performance (see our technical report [4, Sec. 10]).

Internally, the model uses three distributions: the hit and eviction distributions (already discussed) and the age distribution of cached lines (i.e., the probability that a randomly selected line has age a). These distributions are interdependent and related to one another by simple equations. Each incorporates a particular model input and conveys its constraints: *(i)* the age distribution incorporates the cache size, and constrains modeled capacity; *(ii)* the hit distribution incorporates the access stream and constrains when lines can hit; and *(iii)* the eviction distribution incorporates the replacement policy and constrains how long lines stay in the cache. The model is solved by iterating to a fixed point. When the distributions converge, the hit and eviction distributions accurately describe the cache's behavior.

We build the model in stages. First, we introduce the model on a specific example. Second, we develop the model formulation for LRU. Third, we generalize it to other policies. Finally, we show to solve the model efficiently.

3.1 Example

Table 1 shows the behavior of a 3-line LRU cache on a simple, repeating access pattern. Time (measured in accesses) increases from left to right and wraps around (e.g., time 9 would be identical to 1). Live lines, those that eventually hit, are colored green; dead lines, those that do not, are red.

Request	A	A	B	C	B	D	B	C
Lines	🟩 A	🟥 A	🟥	🟥	🟥	🟥 D	🟥	🟥
	🟩	🟩	🟩 B	🟩	🟩 B	🟩	🟩 B	🟩
	🟩	🟩	🟩	🟩 C	🟩	🟩	🟩	🟩 C
Time	1	2	3	4	5	6	7	8

Table 1: Steady-state behavior of a 3-line LRU cache on a simple, repeating access pattern. Live lines are colored green, and dead lines red.

The first line is live (green) at time 1 because A hits at time 2. However, D evicts A at time 6, so A is dead (red) in 2–5. Similarly, A evicts D at time 1, so D is dead in 6–9. B and C always hit, so they are always live. This divides the cache into *lifetimes*, the intervals from last use until hit or eviction. For example, A's lifetime starting at time 1 ends after a single access when A hits at time 2; this starts a new lifetime that ends after four accesses when D evicts A at time 6.

We can redraw Table 1 showing the ages of each line at each access (Table 2). In steady state, 6 out of 8 accesses are hits. These hits (underlined) come at ages 1 (A once), 2 (B twice), and 4 (C twice and B once). The other 2 accesses are evictions (*italic*), at ages 3 (D) and 4 (A).

Request	A	A	B	C	B	D	B	C
Ages	🟩 1	🟥 1	🟥 2	🟥 3	🟥 *4*	🟥 1	🟥 2	🟥 *3*
	🟩 3	🟩 4	🟩 1	🟩 2	🟩 1	🟩 2	🟩 1	🟩 2
	🟩 2	🟩 3	🟩 4	🟩 1	🟩 2	🟩 3	🟩 4	🟩 1
Time	1	2	3	4	5	6	7	8

Table 2: Ages per line in Table 1 (note hits and *evictions*).

This information lets us compute the hit and eviction distributions, denoted by random variables H and E, respectively. These distributions give the probability that a (randomly selected) access will hit or be evicted at a given age. For example, one quarter of accesses hit at age 2 (B at times 4 and 6), so the hit distribution at age 2 is $P_H(2) = 1/4$. Table 3 summarizes our notation and the most important distributions in the model.

Table 4 gives the three model distributions for the example in Table 1. For the hit and eviction distributions, we turn counts into probabilities by dividing by 8 accesses. Since every lifetime ends in a hit or eviction, the hit and eviction distributions together sum to 1, but separately sum to less than 1. We express the distribution inclusive of both hits and evictions as the *lifetime distribution*, $P_L(a) = P_H(a) + P_E(a)$ (not shown in Table 4). L is the age at which a line is either hit or evicted, i.e. the age when its lifetime ends. Finally, we can compute the age distribution by counting the ages of lines and dividing by 3 lines × 8 accesses = 24.

Symbol	**Meaning**	
$P[y]$	The probability of event y.	
$E[X]$	The expected value of random variable (rv) X.	
$P_X(x)$	The probability that rv X equals x, $P[X=x]$.	
$P_{X,Y}(x,y)$	The joint probability rv X equals x and rv Y equals y.	
$P_X(x \mid y)$	The conditional probability that rv X equals x given y.	
RV	**Meaning**	
D	Reuse distance of an access.	Input
H	Age at which a line hits.	Output
E	Age at which a line is evicted.	
A	Age of a line.	Internal
L	Lifetime of a line (see text).	

Table 3: Notation used in this paper. D is a property of the access stream; A, H, and E of cached lines.

	Age a	1	2	3	4	5	6	7	8	Sum
Distribution	$P_D(a)$	$1/8$	$2/8$	–	$3/8$	–	–	$1/8$	$1/8$	**1**
	$P_A(a)$	$1/3$	$7/24$	$5/24$	$1/6$	–	–	–	–	**1**
	$P_H(a)$	$1/8$	$1/4$	–	$3/8$	–	–	–	–	**$3/4$**
	$P_E(a)$	–	–	$1/8$	$1/8$	–	–	–	–	**$1/4$**

Table 4: Model distributions for the cache in Table 1.

These distributions tell us how the cache performs on this access pattern: its hit rate is the sum over the hit distribution, $3/4$. Moreover, the distributions also say how lines behave. For example, no lines make it past age 4, despite some having reuse distance of 8.

In this example, we have computed the distributions by brute force, but this method is too expensive for our purposes. We instead model the relationships among distributions analytically and solve them by iteration. This example gives the intuition behind the model, but note that *our independence assumptions are accurate only for large caches*. So while our model is accurate on real systems (Sec. 8), some model components are inaccurate on simple examples like Table 1.

4. BASIC MODEL (FOR LRU)

This section presents the model for caches with LRU replacement. We present the complete equations for the age and hit distributions, and the eviction distribution equations for LRU replacement. Sec. 5 extends the eviction distribution to model arbitrary age-based replacement policies.

4.1 Model assumptions

First, we make a few simplifying assumptions that make cache modeling tractable, chiefly about the independence of certain events. These assumptions are motivated by the properties of modern LLCs (Sec. 2.1).

We assume each access has reuse distance d distributed identically and independently according to the distribution $P_D(d)$. This assumption is not exactly correct in practice, but since private caches filter accesses before they reach the LLC, it is a good approximation for large caches [5]. Indeed, the access patterns that cause large errors—highly correlated references—are precisely those that cache well in the private cache levels. In other work, we have shown that this assumption is surprisingly robust, yielding accurate predictions even on access patterns that violate it [6]. Sec. 8 validates our model against real programs at many different cache sizes and shows that it is accurate. In return, this assumption greatly simplifies the probability calculations, allowing a simple model to capture diverse access patterns.

Similarly, we model an idealized, *random-candidates* cache, where replacement candidates are drawn at random from cached lines. This is a good model for modern LLCs, where the replacement candidates form a representative sample of cached lines. The random-candidates model is a direct analog of skew-associative caches or zcaches, but is also a good approximation for hashed, set-associative caches with many ways [44]. Although the accuracy of this assumption depends slightly on the cache array architecture, it is a reasonable simplifying assumption for modern LLCs.

Finally, we assume the victim's age is independent of whether the cache hits or misses. This is sensible because a large cache is typically unaffected by a single candidate. The need for this assumption will become clear below.

4.2 Age distribution

The age distribution is used internally by the model to constrain cache capacity. It is presented first because it is the simplest to compute from the other distributions.

Since ages measure the time since a line was last referenced, a line reaches age a if and only if it is not hit or evicted for at least a accesses. Hence the probability of a line having age a, $\mathrm{P}_A(a)$, is proportional to the fraction of lines that survive at least a accesses in the cache without being referenced. The probability of surviving *exactly* x accesses is given by the lifetime distribution at age x, $\mathrm{P}_L(x) = \mathrm{P}_H(x) + \mathrm{P}_E(x)$, and the probability of surviving *at least* a accesses is the probability over all $x \geq a$. The age distribution $\mathrm{P}_A(a)$ is thus proportional to $\mathrm{P}[L \geq a]$, which determines the age distribution up to a constant factor.

To find the constant factor, note that every access necessarily produces a line of age 1 (whether a hit or miss), since we define ages as the number of accesses since a line was last referenced. Since ages increase upon each access, there is always exactly one line of age 1 in the cache. Hence if the cache has S lines, the age distribution at age 1 is $\mathrm{P}_A(1) = {}^1\!/\!_S$.

Combining the two results, we find the age distribution for a cache of S lines is:

$$\mathrm{P}_A(a) = \frac{\mathrm{P}[L \geq a]}{S} = \sum_{x=a}^{\infty} \frac{\mathrm{P}_H(x) + \mathrm{P}_E(x)}{S} \tag{1}$$

For example, in Sec. 3.1, by counting cells we find that $\mathrm{P}_A(3) = 5/24$. That is, of the twenty-four cells depicted in Table 2, age 3 appears five times. Eq. 1 gives another way to arrive at the same conclusion without counting cells: The probability of a hit or eviction at age 3 or greater is $3/8 + 1/8 + 1/8 = 5/8$ (Table 4). Dividing by $S = 3$ gets $\mathrm{P}_A(3) = 5/24$, as desired. This argument can be generalized to yield an alternative derivation of Eq. 1 and show $\sum_{a=1}^{\infty} \mathrm{P}_A(a) = 1$.

4.3 Hit distribution

We now show how to compute when hits occur for a given access pattern, again assuming the other distributions are known. The hit distribution is perhaps the most important product of the model, since it yields the cache's hit rate.

A line will eventually hit if it is not evicted. Intuitively, a line's hit probability depends on its reuse distance (longer distances have greater opportunity to be evicted) and the cache's eviction distribution (i.e., at what age does the replacement policy evict lines?). Moreover, by definition, a line with reuse distance d must hit at age d if it hits at all. Thus the hit probability is just the reuse distance probability minus the probability of being evicted, or:

$$\begin{aligned} \mathrm{P}_H(d) &= \mathrm{P}_D(d) - \mathrm{P}[\text{evicted}, D = d] \\ &= \mathrm{P}_D(d) - \sum_{a=1}^{d-1} \mathrm{P}_{D,E}(d,a) \end{aligned} \tag{2}$$

It may be tempting to simply subtract the eviction probability below d in the above equation. That is, to say $\mathrm{P}_H(d)$ equals $\mathrm{P}_D(d) \times (1 - \mathrm{P}[E < d])$. This is incorrect; eviction age and reuse distance are not independent.

To proceed, we require the critical insight that candidates of age a look alike to the replacement policy. In other words, the replacement policy does not know candidates' reuse distances, only that lines of age a have reuse distance $d > a$. Since reuse distances are iid, evicting at age a thus only implies that the line's reuse distance was at least a, i.e. $E = a \Rightarrow D > a$. From this insight:

$$\begin{aligned} \mathrm{P}_{D,E}(d,a) &= \mathrm{P}_D\big(d|E = a\big)\ \mathrm{P}_E(a) \\ \text{(Insight)} \qquad &= \mathrm{P}_D\big(d|D > a\big)\ \mathrm{P}_E(a) \\ \text{(Simplify)} \qquad &= \frac{\mathrm{P}_D(d) \cdot \mathrm{P}_E(a)}{\mathrm{P}[D > a]} \end{aligned} \tag{3}$$

This insight relies on reuse distances being (approximately) iid. Hence, Eq. 3 is accurate for the large LLCs seen in real systems (Sec. 8), but it is inaccurate for the small example in Sec. 3.

Finally, we substitute this result into the above equation: Summing over all a below d counts all the lines of reuse distance d that are evicted before hitting. Since the rest hit, subtracting these from $\mathrm{P}_D(d)$ yields the hit probability:

$$\mathrm{P}_H(d) = \mathrm{P}_D(d) \times \left(1 - \sum_{a=1}^{d-1} \frac{\mathrm{P}_E(a)}{\mathrm{P}[D > a]}\right) \tag{4}$$

4.4 Eviction distribution in LRU

The eviction distribution accounts for the replacement process and is the most complex part of the model. It models both the selection of replacement candidates (i.e., associativity) and the selection of a victim among candidates (i.e., the replacement policy). For clarity, we begin by presenting the model for LRU only.

To compute the eviction distribution, we assume that candidates are drawn randomly from the cache, as discussed above. (We do *not* assume victims are selected randomly, just that candidates are.) Among these, LRU simply evicts the oldest, so the eviction probability at age a is the probability that both an access misses and a is the oldest age among candidates. These events are independent by the final assumption above.

Hence to find the eviction probability at age a, we need to know the probabilities that the access misses and the oldest candidate has age a: $\mathrm{P}_E(a) = \mathrm{P}[\text{miss}] \cdot \mathrm{P}_{\text{oldest}}(a)$. The first factor is trivial, since

$$\mathrm{P}[\text{miss}] = 1 - \mathrm{P}[\text{hit}] = 1 - \sum_{a=1}^{\infty} \mathrm{P}_H(a) \tag{5}$$

The challenge lies in finding the distribution of oldest ages, $\mathrm{P}_{\text{oldest}}(a)$. Replacement candidates are drawn randomly from the cache, so each has age identically and independently distributed according to $\mathrm{P}_A(a)$. The oldest age is just the maximum of W iid random variables, where W is the associativity. The maximum of iid random variables is a well-known result [19]: the oldest age among replacement candidates is less than a iff all candidates are of age less than a. Thus given W candidates, $\mathrm{P}[\text{oldest} < a] = \mathrm{P}[A < a]^W$. To get the distribution from the cumulative distribution, differentiate: $\mathrm{P}_{\text{oldest}}(a) = \mathrm{P}[\text{oldest} < a+1] - \mathrm{P}[\text{oldest} < a]$. Altogether, the eviction distribution at age a for LRU is:

$$\begin{aligned} \mathrm{P}_E(a) &= (1 - \mathrm{P}[\text{hit}])\ \mathrm{P}_{\text{oldest}}(a) \qquad (6) \\ &= (1 - \mathrm{P}[\text{hit}]) \big(\mathrm{P}[\text{oldest} < a+1] - \mathrm{P}[\text{oldest} < a]\big) \\ \Rightarrow \mathrm{P}_E(a) &= (1 - \mathrm{P}[\text{hit}]) \Big(\mathrm{P}[A < a+1]^W - \mathrm{P}[A < a]^W\Big) \end{aligned}$$

4.5 Summary

Equations 1, 4, and 6 form the complete model for LRU replacement. The age distribution incorporates the cache size and depends on the hit and eviction distributions. The hit distribution incorporates the access stream and depends on the eviction distribution. The eviction distribution incorporates the cache's associativity and replacement policy (LRU) and depends on the hit and age distributions.

5. OTHER REPLACEMENT POLICIES

We now extend the eviction distribution to support arbitrary age-based policies, like those discussed in Sec. 2.2.

5.1 Ranking functions

To support other policies, we must abstract the replacement policy in a way that can be incorporated into the model. We do so through a *ranking function*, $\mathcal{R} : \text{age} \to \mathbb{R}$, which gives an eviction priority to every age. By convention, higher rank means higher eviction priority.

Ranking functions capture many existing policies. For example, LRU's ranking function is $\mathcal{R}_{\text{LRU}}(a) = a$ (or any other strictly increasing function). This represents LRU because it ensures that older lines will be preferred for eviction. Similarly, a constant ranking function produces random replacement, e.g. $\mathcal{R}_{\text{random}}(a) = 0$. Ranking functions can also capture many high-performance replacement policies.

PDP [16] protects lines up to an age d_p, known as the protecting distance. It prefers to evict lines older than the protecting distance, but if none are available among candidates, it evicts the youngest line. Thus PDP's ranking function decreases up to the protecting distance (d_p), upon which it jumps to a large value and increases thereafter:

$$\mathcal{R}_{\text{PDP}}(a) = \begin{cases} d_p - a & \text{If } a < d_p \\ a & \text{If } a \geq d_p \end{cases} \tag{7}$$

IRGD [51] ranks lines using an analytical formulation based on the reuse distance distribution. IRGD essentially ranks lines by their expected reuse distance, but since in practice very large reuse distances can't be measured, IRGD uses a weighted harmonic mean instead of the conventional, arithmetic mean. This lets it ignore immeasurably large reuse distances, since they have a small impact on the harmonic mean.[1] Its ranking function is:

$$\mathcal{R}_{\text{IRGD}}(a) = \mathrm{P}[D > a] \times \left(\sum_{x=1}^{\infty} \frac{\mathrm{P}_D(a+x)}{a+x} \right)^{-1} \tag{8}$$

Ranking functions thus model any age-based policy, but not all high-performance policies are strictly age-based. Our model can support such policies (e.g., RRIP [23, 54]) with specialized extensions. However, this paper presents the general framework, and we leave extensions to future work.

From the ranking function and age distribution, we can produce a *rank distribution* that gives the probability a line will have a given rank. It is then possible to generalize Eq. 6. While LRU evicts the oldest replacement candidate, in general the cache evicts the maximum rank among candidates.

5.2 Generalized eviction distribution

Generalizing the eviction distribution is a straightforward substitution from "oldest age" in LRU to "maximum rank" in general. If a line of age a is evicted, then the maximum rank among candidates must be $\mathcal{R}(a)$. Additionally, $\mathcal{R}$ may rank several ages identically (i.e., $\mathcal{R}(a) = \mathcal{R}(b)$), so we must ensure that the candidate had age a (not age b).

This consideration is important because, in practice, continuous ranks are quantized in units of Δr, increasing the possibility that several ages map to indistinguishable ranks. For example, if ranks can take values in $[0, 256)$ (e.g., LRU with 8-bit ages), then an efficient model implementation might quantize ranks into regions as $[0,8), [8,16) \ldots [248,256)$. Each region has size $\Delta r = 8$, and many ages may have the "same rank" as far as the model is concerned.

We account for indistinguishable ranks by using the joint distribution of rank and age to avoid double counting:

$$\mathrm{P}_E(a) = (1 - \mathrm{P}[\text{hit}]) \cdot \mathrm{P}_{\text{max rank},A}\big(\mathcal{R}(a), a\big) \tag{9}$$

The joint distribution is in turn:

$$\mathrm{P}_{\text{max rank},A}\big(\mathcal{R}(a), a\big) = \mathrm{P}_{\text{max rank}}\big(\mathcal{R}(a)\big) \cdot \frac{\mathrm{P}_A(a)}{\mathrm{P}_{\text{rank}}\big(\mathcal{R}(a)\big)} \tag{10}$$

$\mathrm{P}_A(a)/\mathrm{P}_{\text{rank}}\big(\mathcal{R}(a)\big)$ is the fraction of ranks belonging to age a in the rank quantum containing $\mathcal{R}(a)$ (roughly its Δr-neighborhood). Multiplying by this fraction eliminates double counting. Eq. 10 should simply be thought as the analog to $\mathrm{P}_{\text{oldest}}(a)$ in LRU.

As in LRU, the challenge lies in finding $\mathrm{P}_{\text{max rank}}(r)$. To start, we compute the *rank distribution* in the cache from the age distribution. Since ranks depend on age, the probability that a line's rank equals r is just the total probability of ages with rank r:

$$\mathrm{P}_{\text{rank}}(r) = \sum_{a:\mathcal{R}(a)=r} \mathrm{P}_A(a) \tag{11}$$

[1] Takagi et al. [51] express IRGD somewhat differently, but the two formulations are equivalent.

Next, the cumulative distribution of maximum rank is computed just as $\text{P}_{\text{oldest}}(a)$ in LRU as the maximum of iid random variables:

$$\text{P}[\max\text{rank} < r] = \text{P}[\text{rank} < r]^W \tag{12}$$

Finally, the distribution of maximum rank is obtained by discrete differentiation [43]:

$$\text{P}_{\max\text{rank}}(r) = \frac{\text{P}[\max\text{rank} < r + \Delta r] - \text{P}[\max\text{rank} < r]}{\Delta r} \tag{13}$$

(In LRU, the oldest age distribution uses $\Delta r = 1$.)

These formulae fill in all the terms to compute the generalized eviction distribution:

$$\text{P}_E(a) = \big(1 - \text{P}[\text{hit}]\big) \times \left(\frac{\text{P}_A(a)}{\text{P}_{\text{rank}}\big(\mathcal{R}(a)\big)}\right) \times \left(\frac{\text{P}\big[\text{rank} < \mathcal{R}(a) + \Delta r\big]^W - \text{P}\big[\text{rank} < \mathcal{R}(a)\big]^W}{\Delta r}\right) \tag{14}$$

5.3 Discussion

The complete cache model is given by the age (Eq. 1), hit (Eq. 4), and eviction (Eq. 14) distributions. These equations describe a cache using an arbitrary, age-based replacement policy. Our model forms a system of equations that describe a valid solution, but does not yield this solution directly.

The implicit nature of our model has benefits. The equations organize the model into logical components. Each distribution is responsible for a specific model input: the age distribution for the cache size, the hit distribution for the access pattern, and the eviction distribution for replacement (both associativity and replacement policy). This makes it easy to adapt the model to new cache architectures. For example, a new replacement policy only requires a new ranking function, and all appropriate changes naturally propagate through the eviction, hit, and age distributions. Likewise, new applications change only the reuse distance distribution.

However the drawback is that, since these equations are not explicit, their solution is not entirely obvious. We solve the system through iteration to a fixed point, discussed next.

6. MODEL SOLUTION

All components of the model are interdependent, and a general, closed-form solution is unlikely to exist. We solve the model by iterating to a fixed point, *simultaneously* solving the three distributions age by age (Algorithm 1). This simultaneous solution tightly couples the solution of each distribution to the others, maintaining their relationships. That is, each distribution is computed from the others as the solution evolves, rather than from the distributions at the last iteration. Only the hit rate and rank distribution are fixed across iterations. We find this tight coupling improves convergence time.

At each iteration, Algorithm 1 solves Eq. 1, Eq. 4 and Eq. 14 for age a in constant time, building on the solution from age $a - 1$. Algorithm 1 uses the following recurrence relation derived from Eq. 1:

$$\text{P}_A(a+1) = \text{P}_A(a) - \frac{\text{P}_H(a) + \text{P}_E(a)}{S} \tag{15}$$

This allows the age distribution to be updated to reflect the hit and eviction distributions as the solution evolves, which in turn influences the solution of the hit and eviction distributions. The hit and eviction distributions are thus constrained, and negative feedback loops are imposed on over-estimation. Sums in other equations are similarly broken across iterations so that each age is solved in constant time. For example, the variable evBelow is the inner sum in Eq. 4.

Algorithm 1. The cache model simultaneously solves the cache's age, hit, and eviction distributions by iteration.

```
Inputs: $S$ - Cache size; $W$ - associativity; $\mathcal{R}$ - ranking function; rdd - reuse distance
    distribution;
Returns: Hit and eviction distributions, hit and evict.
function Model
    age, hit, evict, h' ← Seed                                    ▷ Initialize distributions.
    while not Converged:
        h ← h'                                                  ▷ Hit rate from last iteration.
        h' ← 0                                         ▷ $x'$ is solution of $x$ for this iteration.
        crd ← rdd[1]                                   ▷ Cumulative $D$ probability, $\text{P}[D \leq a]$.
        evBelow ← 0                                    ▷ Prob. line evicted at $D = a$ in Eq. 4.
        age'[1] ← $1/S$
        for $a$ ← 1 to $N$:
            hit'[$a$] ← rdd[$a$] (1 − evBelow)
            evict'[$a$] ← (1−h) maxRankDist[$\mathcal{R}(a)$] $\frac{\text{age}'[a]}{\text{rankDist}[\mathcal{R}(a)]}$
            age'[$a$+1] ← age'[$a$] − (hit'[$a$] + evict'[$a$])/$S$
            h' ← h' + hit'[$a$]
            evBelow ← evBelow + evict'[$a$]/(1 − crd)
            crd ← crd + rdd[$a$+1]
        age, hit, evict ← Average(age', hit', evict')
    return hit, evict
```

We seed the first iteration with sensible but arbitrary parameters (e.g., hit rate of 50%). To avoid oscillating around a stable solution, in each iteration we average the old and new distributions using an exponentially weighted moving average. We have empirically determined that a coefficient of ⅓ yields good performance. We detect convergence when the hit rate stays within a 10^{-3} range for ten iterations. Finally, the model sets a floor of 0 for all probabilities during solution. In practice, Algorithm 1 reliably converges after a few iterations (typically 20-40) on hundreds of thousands of distributions from real workloads (Sec. 8).

While involved, iteration is computationally cheap: in practice, we use and monitor coarse ages (see below) for which N-point distributions with $N \approx$ 64–256 suffice, and each iteration runs in linear time on the size of the distributions. Our C++ implementation is publicly available at `http://people.csail.mit.edu/sanchez`.

6.1 Convergence

Our model is designed with generality in mind, but this comes at the cost of complicating some theoretical properties. Eq. 1, Eq. 4, and Eq. 14 form a non-linear system (particularly Eq. 14) operating in many dimensions (N points per distribution and multiple distributions). Moreover, the model accepts arbitrary N-point vectors as input (the reuse distance distribution and ranking function). Demonstrating the convergence of fixed point iteration for non-linear systems is difficult. Generally, it involves reducing the system to a contraction mapping of some relevant model parameter [24]. Although several intuitive parameters are attractive (e.g., hit rate or modeled cache size), we cannot yet prove a contraction mapping on these parameters in general—indeed, it seems that for some degenerate ranking functions (not those in Sec. 5.1), the model does not converge.

We instead take an empirical approach. We evaluate our model on a diverse suite of real applications and demonstrate its accuracy and utility in that context. Since the model is solved online at regular intervals, our evaluation represents hundreds of thousands of model solutions. Thus we conclude that the model converges on distributions seen in practice.

The model converges to a stable miss rate because there is a "tug-of-war" between the age and eviction distributions. Evictions reduce the age probability (Eq. 15), so if the modeled miss rate is too high, then fewer evictions will occur. This decreases the modeled miss rate at the next iteration, tending towards equilibrium; the converse holds as well. However, we leave rigorous convergence conditions to future work.

6.2 Increased step size

Reuse distances in practice can be quite large, and naïve iteration over all ages would be quite expensive. Moreover, age-by-age iteration is wasteful, since there are large age ranges where few events occur or the event probabilities are roughly constant. Modeling such ranges in detail is unnecessary, since they can be approximated by assuming constant event probabilities throughout the range. This observation allows the solution to take a large step over many ages at once and greatly reduces N, the inner loop iterations in Algorithm 1. Reducing N is important when applying the model online, e.g. as part of a runtime system (Sec. 9).

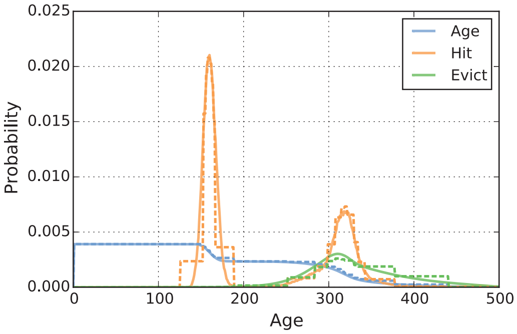


Figure 2: Model solution with increased step size on a synthetic benchmark. Solid lines show a full-detail solution; dashed lines show a solution with $N = 32$.

For example, Fig. 2 shows a solution of all three model distributions for a synthetic benchmark (solid lines). These distributions are *coarsened* by increasing the step size (dashed lines), producing a good approximation with much less computation. Indeed, Fig. 2 shows that $N = 32$ is sufficient to model this access pattern, even though ages go up to 500. This is possible because there are large regions (e.g., ages 0-100) where few events occur. There is no reason to model these regions in great detail. Instead, we adaptively divide ages into regions, modeling regions of high activity at fine granularity and others at coarse granularity.

We then model the total probability within each coarsened region. That is, rather than computing the hit probability at a single age (e.g., $\mathrm{P}[H = a]$), we compute the hit probability over several ages (e.g., $\mathrm{P}[a \leq H < b]$). Remarkably, the model equations are basically unchanged by coarsening. For example, if regions are split at ages $a_1, a_2, a_3, \ldots$ then the coarsened hit equation is:

$$\mathrm{P}[a_i \leq H < a_{i+1}] \approx \qquad (16)$$
$$\mathrm{P}[a_i \leq D < a_{i+1}] \times \left(1 - \sum_{j=1}^{i-1} \frac{\mathrm{P}\big[a_j \leq E < a_{j+1}\big]}{\mathrm{P}\big[D > a_j\big]} \right)$$

This equation is identical in form to the fine-grain hit equation (Eq. 4), except now operating on regions rather than individual ages. The same pattern holds for other equations [4, Appendix A].

The model is unchanged under coarsening for a deep reason. Our model can be relaxed into a system of differential equations through some simple transformations [6]. From this perspective, coarsening is essentially a numerical solution of the system using an adaptive step size [43]. Such solutions preserve the form of the original equations.

Another important question is how to choose the age regions. The choice must balance two competing goals: modeling regions with high activity at fine granularity, while modeling other ages in sufficient detail for the model to converge. We address this in two steps. First, we divide all ages evenly into $N/2$ regions. For example, with 8-bit ages and $N = 64$, we first create the 32 regions divided at ages: $0, 8, 16 \ldots 256$. Second, we further divide these regions $N/2$ times to try to equalize the probability of hits and evictions in each region. We sort regions by their probability of hits and evictions, and recursively divide the largest in equal-probability halves $N/2$ times. We find this procedure chooses regions that yield efficient and accurate solutions.

7. IMPLEMENTATION

We now describe how to integrate our cache model into a full system which we evaluate in simulation. In our validation (Sec. 8) and case study (Sec. 9), the model is used to dynamically model or reconfigure the cache, as shown in Fig. 3. A lightweight hardware monitor samples a small fraction of accesses and produces the application's reuse distance distribution. Periodically (e.g., every 100 ms), a software runtime models the cache's behavior from the sampled reuse distance distribution, which is then used to predict the cache's behavior over the next interval.

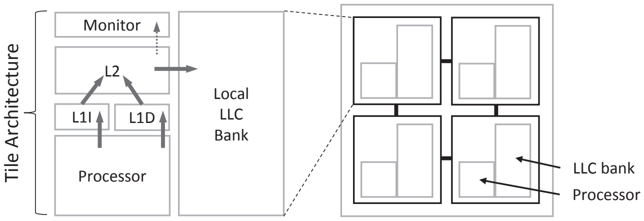


Figure 3: An example implementation of our model. A lightweight hardware monitor on each tile samples a small fraction of LLC accesses. Our model runs periodically in software.

This configuration represents just one use case for the model; it can also be used to model cache behavior offline. However, Fig. 3 is the most demanding use case, since it imposes the most stringent run-time requirements and the model must contend with sampling error.

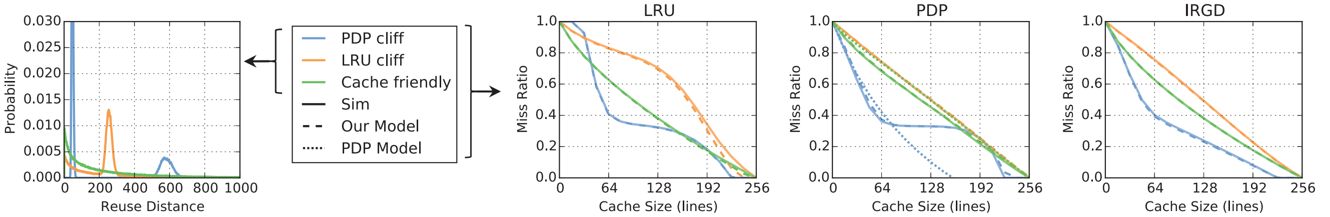


Figure 4: The cache model on three synthetically generated traces driving small caches using LRU, PDP, and IRGD replacement. Simulation results are shown as solid lines; model predictions as dashed lines; PDP's model as dotted lines.

7.1 Application profiling

Our model works with several low-overhead, hardware monitoring schemes. For instance, PDP [16] proposes a FIFO that samples a small fraction of accesses and, when an access hits in the FIFO, records its depth in the FIFO as the reuse distance. Other monitors record stack distance and could be adapted to work with our model. For example, utility monitors [41] are a small, tag-only LRU cache that record stack distance. Geometric monitors [7] extend utility monitors to efficiently monitor very large stack distances. Stack distances can then be approximately translated into reuse distances in software [47].

In our implementation, we add a small, tagged, LRU array that samples 1% of accesses through an H3 hash function [10]. The monitor thus behaves like a large cache [5]. The monitor keeps a global timestamp counter that is incremented on every sampled access. Each monitor entry holds a 16 b hashed, partial tag and a coarsened timestamp, which stores the value of the global timestamp divided by a constant $A = 32$. Upon a hit in the monitor, we compute the reuse distance as the (coarsened) global timestamp minus the line's timestamp, and then increment a counter corresponding to the reuse distance. Thus the counter records a histogram of coarsened reuse distances. Monitors impose small overheads, roughly 1% of LLC area, that scale in proportion to cache size.

Finally, other schemes can sample the reuse distance distribution without adding hardware. Software-only schemes can sample access patterns, e.g. by injecting page faults [57]. Offline profiling can record the access trace, e.g. through compiler hooks [15] or dynamic binary translation [52]. These schemes should enable our model when hardware support is unavailable, although we have not evaluated them in detail.

7.2 Overheads

Our model requires modest run-time overheads and small monitoring overheads, similar to prior schemes. The model takes only a few arithmetic operations per age region per iteration (~25). With $N = 128$ and 30 iterations on average, the model completes in ~100 K arithmetic operations. Since the model runs infrequently (e.g., every 100 ms), this overhead is small and constant per core in most applications (see Sec. 9). If this overhead is too large, N can be reduced or the reconfiguration interval can be increased, typically at little performance loss ([3, 7], Fig. 6). Alternatively, the model can be solved in the background with low-priority threads that interfere minimally with active applications [25]. Finally, computation can be reduced by specializing the model to particular ranking functions.

Cores	Westmere-like OOO [46], 2.4 GHz
L1 caches	32 KB, 8-way set-associative, split D/I, 1-cycle latency
L2 caches	128 KB priv. per-core, 8-way set-assoc, inclusive, 6-cycle
L3 cache	Shared, non-inclusive, 20-cycle; 16-way, hashed set-assoc
Coherence	MESI, 64 B lines, no silent drops; sequential consistency
Memory	200 cycles, 12.8 GBps/channel, 1 channel

Table 5: Configuration of the simulated system.

8. VALIDATION

We now validate our model on synthetic and real benchmarks, showing that it is accurate over diverse replacement policies, access patterns, and cache sizes.

8.1 Synthetic

Fig. 4 compares the model against simulation of synthetic traces. These experiments demonstrate the model's accuracy in an environment that largely satisfies its assumptions (Sec. 4.1). We simulate a small cache that randomly selects replacement candidates; random-candidates is an idealized model of associativity [44] that matches model assumptions. Each trace is pseudorandomly generated to produce the desired reuse distance distribution.

Each trace represents a different access mix: one is cache-friendly, and two expose cliffs in LRU and PDP, respectively. Their reuse distance distributions are shown on the left. On the right there is one graph per replacement policy. On each graph, simulation results are shown (solid lines) along with model predictions (dashed lines). The PDP graph also includes the predictions of PDP's analytical model (dotted lines).

Our model is accurate on every configuration. The dashed lines are often not visible because model predictions are indistinguishable from simulation results. By contrast, PDP's model exhibits significant error from simulation, failing to distinguish between the LRU-cliff and cache-friendly patterns, and mispredicting the PDP cliff badly.

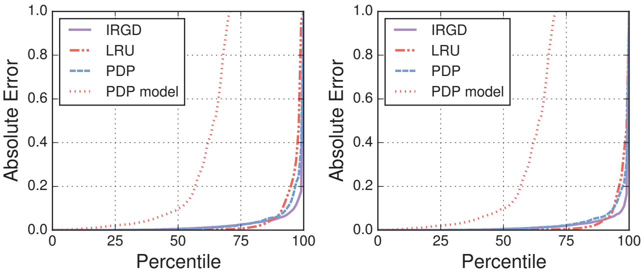
8.2 Execution-driven

These results carry over to real benchmarks on a full system, where model assumptions only approximately hold.

Methodology: We use zsim [46] to evaluate our model. We perform execution-driven simulation of SPEC CPU2006 benchmarks on OOO cores using 16-way hashed, set-associative last-level caches and parameters given in Table 5. We run each benchmark for 10 B instructions after fast-forwarding 10 B, and we perform enough runs to achieve 95% confidence intervals $\leq$1%. All results hold for skew-associative caches [48], zcaches [44], and for systems with prefetching. This methodology also applies to later case studies.

We evaluate a large range of cache sizes, from 128 KB to 128 MB, and solve the model every 250 K accesses from the sampled reuse distance distribution in that interval. This yields many samples—over 400 K model solutions in all.

Results: Fig. 5 summarizes results across all SPEC CPU2006 applications and across all cache sizes from 128 KB to 128 MB. It shows the distribution of modeling error (|predicted hit rate − actual hit rate|) in each 250 K-access interval for LRU, PDP, and IRGD. The figure thus summarizes model accuracy on many diverse cache behaviors (>400 K solutions).



(a) 128-point solution.

(b) Full solution.

Figure 5: Model error distribution over 250 K-access intervals. Our model is accurate, while PDP's model (dotted line) has large error.

We show results *(a)* for coarsened solutions with $N = 128$ (Sec. 6.2) and *(b)* full solutions. For 128-point solutions, median error is 0.1%/0.1%/0.6% for LRU/PDP/IRGD, respectively; mean error is 3.3%/3.7%/2.2%; and 90th percentile error is 7.5%/9.9%/6.1%. For full solutions, median error is 0.1%/1.1%/0.6%; mean error is 2.2%/3.5%/1.9%; and 90th percentile error is 4.8%/9.9%/5.5%.

Overall, the model is accurate, and there is modest error from coarsening solutions. Fig. 6 shows the mean and 90th percentile error for different values of N. (Median error is negligible in all cases.) Our model is fairly insensitive to coarsening, although reducing N below 128 noticeably degrades accuracy (especially on LRU). Skew-associative LLCs improve model accuracy even further.

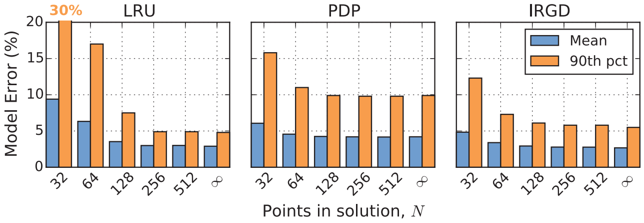
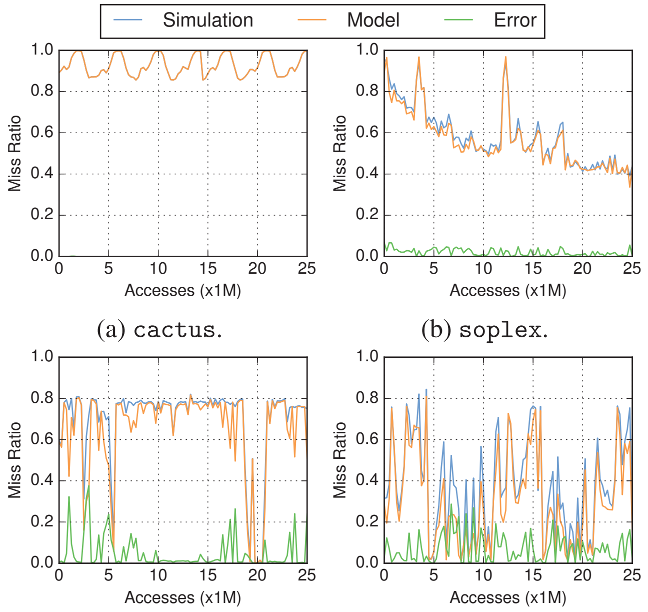


Figure 6: Sensitivity of model to step size for LRU, PDP, and IRGD, measured by the number of steps, N (Sec. 6.2). $N = \infty$ means full, age-by-age solution.

It is important to emphasize that these results are for 250 K-access *intervals*—it is *not* the case that 10% of *benchmarks* have error above 7.5%/9.9%/6.1%. Rather, 10% of intervals have this error. The distinction is critical. Many benchmarks have an unstable access pattern, and their hit rate changes rapidly between intervals. Fig. 7 shows four representative SPEC CPU2006 apps on a 1 MB, LRU LLC, plotting miss ratio over the first 25 M LLC accesses with $N = 128$.[2] Most apps are unstable; gcc's and hmmer's hit rates fluctuate wildly across intervals. Our model does a good job of tracking these fluctuations. (Recall that the model only uses the application's reuse distance distribution; it does not observe the cache's hit rate.) However, since benchmarks are unstable, reuse distance distributions are not representative of equilibrium behavior, and model error is large in some intervals.



(a) cactus.

(b) soplex.

(c) hmmer.

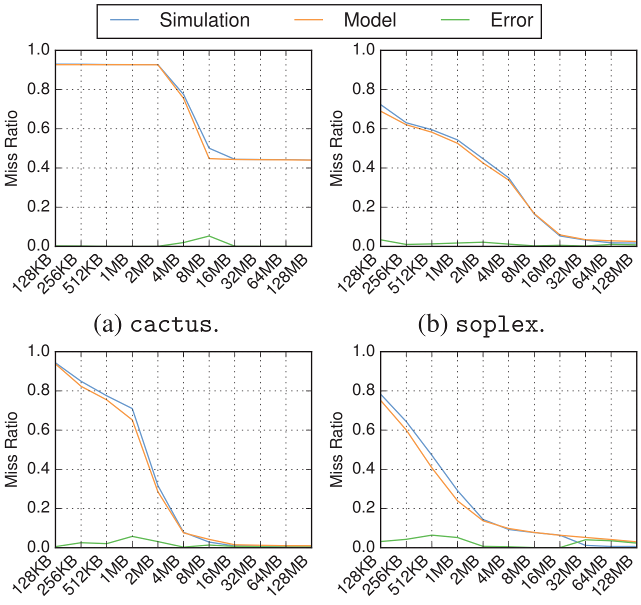
(d) gcc.

Figure 7: Trace of simulated and predicted miss ratio for the first 25 M LLC accesses on a 1 MB, LRU LLC. Some benchmarks are unstable over time, introducing model error on some intervals.

This model error tends to average out in the long-run, and indeed our model is quite accurate at predicting each benchmark's miss curve over its entire execution (Fig. 8, next page). For 128-point solutions, the mean model error over 10 B instructions is 1.9%/2.7%/1.1% for LRU/PDP/IRGD, respectively, while the 90th percentile error is 4.7%/6.7%/3.1%. Results for full solutions are similarly reduced, with long-run mean error of 1.2%/2.7%/1.0% and 90th percentile error of 3.3%/6.7%/2.9%. Hence, the model error presented in Fig. 5 is a conservative assessment of our model's accuracy.

Ultimately, the cases where we observe significant model error are on the benchmarks mcf, lbm, and Gems when using LRU or PDP replacement. Essentially, these benchmarks do not satisfy the model's assumption of iid reuse distances, causing the model to mispredict the "cliff" in the miss curve, e.g. predicting a drop in the miss rate at 1 MB instead of at 2 MB for Gems. This error can be removed at modest additional

[2] Model behavior on PDP and IRGD is essentially identical. We omitted these results due to space constraints.



(a) cactus.

(b) soplex.

(c) hmmer.

(d) gcc.

Figure 8: Simulated and predicted miss ratio over 10 B instructions vs. LRU LLC size. Our model is accurate over the entire execution of each benchmark.

complexity by breaking accesses into multiple classes, as described in our technical report [4]. Other than these cases, we find that the model is accurate across the vast majority of benchmarks and cache sizes. The model's accuracy thus compares well with prior modeling work on LRU [47]. We find that in practice the model is accurate and useful at predicting cache behavior, as we now illustrate in a case study.

9. CASE STUDY: CACHE PARTITIONING

In this section, we show how to apply the model to improve shared cache performance with high-performance replacement policies. Recall from Sec. 2.4 that partitioning has many benefits: improving shared cache performance [41], quality of service [26], fairness [40], security [39], etc.

Partitioning algorithms use each application's *miss curve* to size partitions. Miss curves allow the partitioning algorithm to choose sizes that minimize misses. (The optimal solution is NP-complete, but good approximations exist [3, 37, 41].)

In principle, cache partitioning is complementary to high-performance cache replacement: cache partitioning allocates capacity among applications, and the replacement policy then maximizes each application's hit rate on its partition. However, cache partitioning is only effective when miss curves are available—otherwise, software cannot predict the effect of different partition sizes, and therefore cannot size them to achieve system objectives [5].

Our model resolves this conflict, enabling many optimizations on systems with high-performance replacement policies. It can model high-performance replacement policies at arbitrary cache sizes, and thus predicts the replacement policy's miss curve. These miss curves are given to the partitioning algorithm, which can then choose partition sizes that maximize performance (or achieve other objectives).

Methodology: We evaluate the performance of a shared

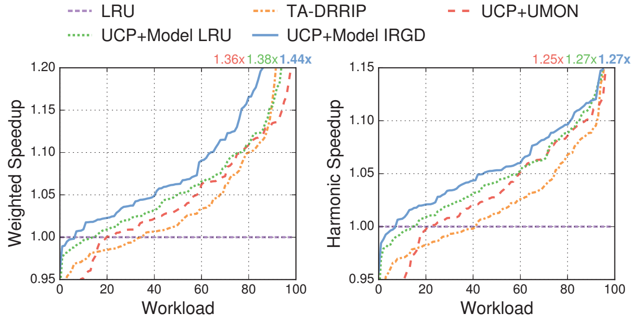


Figure 9: Speedup for 100 random mixes of SPEC CPU2006 apps. Our model enables partitioning of IRGD, improving performance over the state-of-the-art.

4 MB LLC on a 4-core system running 100 random mixes of SPEC CPU2006 applications. We use a fixed-work methodology, similar to FIESTA [21], with each application fast-forwarded 20 B instructions and run for at least 1 B instructions. We only use the first 1 B instructions of execution to compare performance. This case study thus represents over 400 billion instructions of real programs.

We compare five schemes: *(i)* unpartitioned LRU, *(ii)* a representative thread-aware high-performance policy (TA-DRRIP [23]), LRU with utility-based cache partitioning [41] and *(iii)* hardware LRU monitors (UCP+UMON) or *(iv)* our model (UCP+Model LRU), and *(v)* a high-performance policy (IRGD) with utility-based cache partitioning (UCP+Model IRGD) and our model.

Both UCP schemes partition the cache every 100 M cycles using 16-point miss curves per application. For UCP, this means a 16-way utility monitor, which adds small hardware overheads [41]. For UCP+IRGD, this means 16 model solutions. Multiple sizes can be solved independently, so a vectorized implementation can produce a full miss curve in 1.4 Mcycles (using 32-bit floating-point values and 256-bit vectors), imposing a 0.6% runtime overhead.

Results: Fig. 9 shows the distribution of weighted and harmonic speedup over the 100 mixes [18], normalized to unpartitioned LRU. TA-DRRIP and UCP+UMON both outperform the LRU baseline respectively by 4.5% on gmean weighted speedup, and by 2.4%/3.4% on gmean harmonic speedup.

Meanwhile, UCP+Model LRU improves performance gmean weighted speedup by 6.5% and gmean harmonic speedup by 5.1%, outperforming the hardware monitor. We conjecture this is because conflict misses affect UMON accuracy with small partitions, while our model does not suffer this effect.

Finally, UCP+Model IRGD improves by gmean weighted speedup by 10.2% and and gmean harmonic speedup by 6.1%. Our model combines the single-stream benefits of high-performance cache replacement and the shared-cache benefits of partitioning, outperforming the state-of-the-art. These results show that our model is general-purpose, but performs as well as an specialized monitor on LRU.

In a technical report [4], we present a further case study that uses the model to improve IRGD's performance.

10. CONCLUSION

We have presented a cache model for modern LLCs with high-performance replacement policies. Our model is motivated by observations of modern cache architecture that allow us to abstract away details of array organization and focus on modeling the replacement policy. As a result, we capture a broad class of policies at relatively low complexity. We have presented an efficient implementation of the model and thoroughly evaluated its accuracy and implementation tradeoffs. Finally, we showed how to use the model to improve cache performance over state-of-the-art techniques.

Acknowledgments

We thank the anonymous reviewers as well as Harshad Kasture, Po-An Tsai, Mark Jeffrey, Suvinay Subramanian, and Joel Emer for their helpful feedback. This work was supported in part by NSF grant CCF-1318384 and a grant from the Qatar Computing Research Institute.

11. REFERENCES

[1] A. Agarwal, J. Hennessy, and M. Horowitz, "An analytical cache model," *ACM TOCS*, vol. 7, no. 2, 1989.

[2] D. H. Albonesi, "Selective cache ways: On-demand cache resource allocation," in *MICRO-32*, 1999.

[3] N. Beckmann and D. Sanchez, "Jigsaw: Scalable software-defined caches," in *PACT-22*, 2013.

[4] N. Beckmann and D. Sanchez, "A cache model for modern processsors," MIT, Tech. Rep. MIT-CSAIL-TR-2015-011, 2015.

[5] N. Beckmann and D. Sanchez, "Talus: A simple way to remove cliffs in cache performance," in *HPCA-21*, 2015.

[6] N. Beckmann and D. Sanchez, "Cache calculus: Modeling caches through differential equations," *Computer Architecture Letters*, 2016.

[7] N. Beckmann, P.-A. Tsai, and D. Sanchez, "Scaling distributed cache hierarchies through computation and data co-scheduling," in *HPCA-21*, 2015.

[8] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," *IBM Sys. J.*, vol. 5, no. 2, 1966.

[9] E. Berg and E. Hagersten, "StatCache: A probabilistic approach to efficient and accurate data locality analysis," in *ISPASS*, 2004.

[10] J. L. Carter and M. N. Wegman, "Universal classes of hash functions (extended abstract)," in *Proc. STOC-9*, 1977.

[11] D. Chiou, P. Jain, L. Rudolph *et al.*, "Application-specific memory management for embedded systems using software-controlled caches," in *DAC-37*, 2000.

[12] H. Cook, M. Moreto, S. Bird *et al.*, "A hardware evaluation of cache partitioning to improve utilization and energy-efficiency while preserving responsiveness," in *ISCA-40*, 2013.

[13] S. Das, T. M. Aamodt, and W. J. Dally, "Reuse distance-based probabilistic cache replacement," *ACM TACO*, vol. 12, no. 4, 2015.

[14] C. Delimitrou and C. Kozyrakis, "Paragon: Qos-aware scheduling for heterogeneous datacenters," in *ASPLOS-XVIII*, 2013.

[15] C. Ding and Y. Zhong, "Predicting whole-program locality through reuse distance analysis," in *PLDI*, 2003.

[16] N. Duong, D. Zhao, T. Kim *et al.*, "Improving cache management policies using dynamic reuse distances," in *MICRO-45*, 2012.

[17] D. Eklov and E. Hagersten, "StatStack: Efficient modeling of LRU caches," in *ISPASS*, 2010.

[18] S. Eyerman and L. Eeckhout, "System-level performance metrics for multiprogram workloads," *IEEE Micro*, vol. 28, no. 3, 2008.

[19] C. M. Grinstead and J. L. Snell, *Introduction to probability*. American Mathematical Soc., 1998.

[20] F. Guo, Y. Solihin, L. Zhao *et al.*, "A framework for providing quality of service in chip multi-processors," in *MICRO-40*, 2007.

[21] A. Hilton, N. Eswaran, and A. Roth, "FIESTA: A sample-balanced multi-program workload methodology," in *Proc. MoBS*, 2009.

[22] S. Jahagirdar, V. George, I. Sodhi *et al.*, "Power management of the third generation Intel Core micro architecture formerly codenamed Ivy Bridge," in *Hot Chips*, 2012.

[23] A. Jaleel, K. Theobald, S. Steely *et al.*, "High performance cache replacement using re-reference interval prediction," in *ISCA-37*, 2010.

[24] M. Javidi, "Iterative methods to nonlinear equations," *Applied Mathematics and Computation*, vol. 193, no. 2, 2007.

[25] M. Kamruzzaman, S. Swanson, and D. M. Tullsen, "Inter-core prefetching for multicore processors using migrating helper threads," in *ASPLOS-XVI*, 2011.

[26] H. Kasture and D. Sanchez, "Ubik: Efficient cache sharing with strict QoS for latency-critical workloads," in *ASPLOS-XIX*, 2014.

[27] G. Keramidas, P. Petoumenos, and S. Kaxiras, "Cache replacement based on reuse-distance prediction," in *ICCD*, 2007.

[28] G. Keramidas, P. Petoumenos, and S. Kaxiras, "Where replacement algorithms fail: A thorough analysis," in *Proc. CF-7*, 2010.

[29] S. M. Khan, Y. Tian, and D. A. Jimenez, "Sampling dead block prediction for last-level caches," in *MICRO-43*, 2010.

[30] S. M. Khan, Z. Wang, and D. A. Jiménez, "Decoupled dynamic cache segmentation," in *HPCA-18*, 2012.

[31] M. Kharbutli, K. Irwin, Y. Solihin *et al.*, "Using prime numbers for cache indexing to eliminate conflict misses," in *HPCA-10*, 2004.

[32] N. Kurd, S. Bhamidipati, C. Mozak *et al.*, "Westmere: A family of 32nm IA processors," in *ISSCC*, 2010.

[33] H. Lee, S. Cho, and B. R. Childers, "CloudCache: Expanding and shrinking private caches," in *HPCA-17*, 2011.

[34] R. Manikantan, K. Rajan, and R. Govindarajan, "Probabilistic shared cache management (PriSM)," in *ISCA-39*, 2012.

[35] J. Mars, L. Tang, R. Hundt *et al.*, "Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations," in *MICRO-44*, 2011.

[36] R. L. Mattson, J. Gecsei, D. R. Slutz *et al.*, "Evaluation techniques for storage hierarchies," *IBM Sys. J.*, vol. 9, no. 2, 1970.

[37] M. Moreto, F. J. Cazorla, A. Ramirez *et al.*, "FlexDCP: A QoS framework for CMP architectures," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 2, 2009.

[38] A. Mukkara, N. Beckmann, and D. Sanchez, "Whirlpool: Improving dynamic cache management with static data classification," in *ASPLOS-XXI*, 2016.

[39] D. Page, "Partitioned cache architecture as a side-channel defence mechanism," *IACR Cryptology ePrint archive*, no. 2005/280, 2005.

[40] A. Pan and V. S. Pai, "Imbalanced cache partitioning for balanced data-parallel programs," in *MICRO-46*, 2013.

[41] M. Qureshi and Y. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *MICRO-39*, 2006.

[42] M. Qureshi, A. Jaleel, Y. Patt *et al.*, "Adaptive insertion policies for high performance caching," in *ISCA-34*, 2007.

[43] L. Richard and J. Burden, "Douglas faires, numerical analysis," 1988.

[44] D. Sanchez and C. Kozyrakis, "The ZCache: Decoupling ways and associativity," in *MICRO-43*, 2010.

[45] D. Sanchez and C. Kozyrakis, "Vantage: Scalable and efficient fine-grain cache partitioning," in *ISCA-38*, 2011.

[46] D. Sanchez and C. Kozyrakis, "ZSim: Fast and accurate microarchitectural simulation of thousand-core systems," in *ISCA-40*, 2013.

[47] R. Sen and D. A. Wood, "Reuse-based online models for caches," in *Proc. SIGMETRICS*, 2013.

[48] A. Seznec, "A case for two-way skewed-associative caches," in *ISCA-20*, 1993.

[49] M. Shah, J. Barren, J. Brooks *et al.*, "UltraSPARC T2: A highly-treaded, power-efficient, SPARC SOC," in *ISSCC*, 2007.

[50] S. Srikantaiah, R. Das, A. K. Mishra *et al.*, "A case for integrated processor-cache partitioning in chip multiprocessors," in *SC09*, 2009.

[51] M. Takagi and K. Hiraki, "Inter-reference gap distribution replacement: an improved replacement algorithm for set-associative caches," in *ICS'04*, 2004.

[52] D. K. Tam, R. Azimi, L. B. Soares *et al.*, "RapidMRC: Approximating L2 miss rate curves on commodity systems for online optimizations," in *ASPLOS-XIV*, 2009.

[53] H. Wong, "Intel Ivy Bridge cache replacement policy, http://blog.stuffedcow.net/2013/01/ivb-cache-replacement/," 2013.

[54] C.-J. Wu, A. Jaleel, W. Hasenplaugh *et al.*, "SHiP: Signature-based hit predictor for high performance caching," in *MICRO-44*, 2011.

[55] M.-J. Wu, M. Zhao, and D. Yeung, "Studying multicore processor scaling via reuse distance analysis," in *ISCA-40*, 2013.

[56] X. Zhang, E. Tune, R. Hagmann *et al.*, "CPI2: CPU performance isolation for shared compute clusters," in *EuroSys*, 2013.

[57] P. Zhou, V. Pandey, J. Sundaresan *et al.*, "Dynamic tracking of page miss ratio curve for memory management," in *ASPLOS-XI*, 2004.