Computer Science and Artificial Intelligence Laboratory

Technical Report

MIT-CSAIL-TR-2015-034 December 19, 2015

# Bridging Theory and Practice in Cache Replacement

Nathan Beckmann and Daniel Sanchez

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

# Bridging Theory and Practice in Cache Replacement

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

## ABSTRACT

Much prior work has studied processor cache replacement policies, but a large gap remains between theory and practice. The optimal policy (MIN) requires unobtainable knowledge of the future, and prior theoretically-grounded policies use reference models that do not match real programs. Meanwhile, practical policies are designed empirically. Lacking a strong theoretical foundation, they do not make the best use of the information available to them.

This paper bridges theory and practice. We propose that practical policies should replace lines based on their *economic value added* (EVA), the difference of their expected hits from the average. We use Markov decision processes to show that EVA is optimal under some reasonable simplifications.

We present an inexpensive, practical implementation of EVA and evaluate it exhaustively over many cache sizes. EVA outperforms prior practical policies and saves area at iso-performance. These results show that formalizing cache replacement yields practical benefits.

## 1. INTRODUCTION

Last-level caches consume significant resources, typically over 50% of chip area [18], so it is crucial to manage them efficiently. Prior work has approached cache replacement from both theoretical and practical standpoints. Unfortunately, there is *a large gap between theory and practice*.

From a theoretical standpoint, the optimal replacement policy is Belady's MIN [6, 19], which evicts the candidate referenced furthest in the future. But MIN's requirement for perfect knowledge of the future makes it impractical. In practice, policies must cope with uncertainty, never knowing exactly when candidates will be referenced.

Theoretically-grounded policies account for uncertainty by using a simplified, statistical model of the reference stream in which the optimal policy can be found analytically. We call such policies *statistically optimal*. The key challenge is defining a statistical model that captures enough information about the access stream to make good replacement decisions, yet is simple enough to analyze. Unfortunately, prior statistically optimal policies rely on the independent reference model (IRM), which is inadequate for processor caches. The IRM assumes that candidates are referenced independently with static probabilities. The IRM-optimal policy is to evict the candidate with the lowest reference probability [1]. Though useful in other areas (e.g., web caches [3]), the IRM is inadequate for processor caches because it assumes that reference probabilities do not change over time.

Instead, replacement policies for processor caches are designed empirically, using heuristics based on observations of common-case access patterns [10, 11, 12, 13, 26, 30, 33]. We observe that, unlike the IRM, these policies do not assume static reference probabilities. Instead, they *exploit dynamic behavior* through various aging mechanisms. While often effective, high-performance policies employ many different heuristics and, lacking a theoretical foundation, it is unclear if any are taking the right approach. Each policy performs well on particular programs, yet no policy dominates overall, suggesting that these policies are not making the best use of available information.

In this paper, we seek to bridge theory and practice in cache replacement with two key contributions. First, we use planning theory to show that the correct approach is to replace candidates by their *economic value added* (EVA); i.e., how many more hits one expects from each candidate vs. the average. Second, we design a practical implementation of this policy and show it outperforms existing policies.

**Contributions:** This paper contributes the following:

- We discuss the two main tradeoffs in cache replacement: hit probability and cache space (Sec. 3), and describe how EVA reconciles them with a single intuitive metric (Sec. 4).
- We formulate cache replacement as a Markov decision process (MDP), and show that EVA follows from MDP theory, though an exact solution is impractical. We describe a simple memory reference model that we use to arrive at our implementation (Sec. 5).
- We present a practical implementation of EVA, which we have synthesized in a 65 nm commercial process. EVA adds 1.3% area overhead on a 1 MB cache vs. SHiP (Sec. 6).
- We evaluate EVA against prior high-performance policies on SPEC CPU2006 and OMP2012 over many cache sizes (Sec. 7). EVA reduces LLC misses over existing policies at equal area, closing 57% of the gap from random replacement to MIN vs. 47% for SHiP [33], 41% for DRRIP [11], and 42% for PDP [10]. *Fewer misses translate into large area savings—EVA matches SHiP's performance with gmean 8% less total cache area.*
- We show that EVA is a general principle of optimal replacement through a case study on compressed caches, where EVA also outperforms the best empirical policy (Sec. 8).

These contributions show that formalizing cache replacement yields practical benefits. EVA is the first rigorous study of cache replacement with dynamically changing reference probabilities, and is consequently the first statistically optimal policy to outperform empirical policies. Our implementation is inspired by iterative MDP algorithms, and is the first adaptive policy that does not require set sampling or auxiliary tagged monitors. Moreover, beyond our particular design, we expect replacement by economic value added (EVA) to be useful in the design of future high-performance policies.

## 2. BACKGROUND

In practice, replacement policies must cope with *uncertainty*, never knowing precisely when a candidate will be

referenced. The challenge is uncertainty itself, since with complete information the optimal policy is simple: evict the candidate that is referenced furthest in the future (MIN [6,19]).

Broadly speaking, prior work has taken two approaches to replacement under uncertainty. On the one hand, architects can develop a probabilistic model of how programs reference memory and then solve for the optimal replacement policy within this reference model. On the other hand, architects can observe programs' behaviors and find best-effort heuristics that perform well on common access patterns.

These two approaches are complementary: theory yields insight into how to approach replacement, which is used in practice to design policies that perform well on real applications at low overhead. For example, the most common approach to replacement under uncertainty is to predict when candidates will be referenced and evict the candidate that is *predicted* to be referenced furthest in the future. This long-standing approach takes inspiration from theory (i.e., MIN) and has been implemented to great effect in recent empirical policies [11, 13]. (Sec. 3 shows that this strategy is suboptimal, however.) Yet despite the evident synergy between theory and practice, the vast majority of research has been on the empirical side; theory dates from the early 1970s [1].

**Replacement theory:** The challenge for statistically optimal policies is to define a reference model that is simple enough to analyze, yet sufficiently accurate to yield useful insights. In 1971, Aho et al. [1] studied page replacement within the *independent reference model* (IRM), which assumes that pages are accessed non-uniformly with known probabilities. They model cache replacement as a Markov decision process, and show that the optimal policy, $A_0$, is to evict the page with the lowest reference probability.

Aho et al. is the work most closely related to ours, but we observe that the independent reference model is a poor fit for processor caches. Like Aho et al., we formalize cache replacement as an MDP, but we use a reference model that tractably captures dynamic behavior (Sec. 5). Capturing dynamic behavior is crucial for good performance. Processor caches are accessed by few threads, so their references tend to be highly correlated. But the IRM assumes static, uncorrelated references, losing this crucial information.

For example, consider a program that scans over a 100 K-line array. Since each address is accessed once every 100 K accesses, each has the same "reference probability". Thus the IRM-optimal policy, which evicts the candidate with the lowest reference probability, cannot distinguish among lines and would replace them at random. In fact, the optimal replacement policy is to protect a fraction of the array so that some lines age long enough to hit. Doing so can significantly outperform random replacement, e.g. achieving 80% hit rate with a 80 K-line cache. But protection works only because of dynamic behavior: a line's reference probability *increases* as it ages. The independent reference model does not capture such information. Clearly, a new model is needed.

**Replacement policies:** Many high-performance policies try to emulate MIN through various heuristics. DIP [26] avoids thrashing by inserting most lines at low priority. SDBP [15] uses the PC to predict which lines are unlikely to be reused. RRIP [11, 33] and IbRDP [13] try to predict candidates' time until reference. PDP [10] protects lines from eviction for a fixed number of accesses. And IRGD [30] computes a statistical cost function from sampled reuse distance histograms. Without a theoretical foundation, it is unclear if any of these policies takes the right approach. Indeed, no policy dominates across benchmarks (Sec. 7), suggesting that they are not making the best use of available information.

We observe two relevant trends in recent research. First, most empirical policies exploit dynamic behavior by using the candidate's age (the time since it was last referenced) to select a victim. For example, LRU prefers recently used lines to capture temporal locality; RRIP [11, 33] predicts a longer re-reference interval for older candidates; PDP [10] protects candidates from eviction for a fixed number of accesses; and IRGD [30] uses a heuristic function of ages.

Second, recent high-performance policies adapt themselves to the access stream to varying degrees. DIP [26] detects thrashing with set dueling. DRRIP [11] inserts lines at medium priority, preferring lines that have been reused, and avoids thrashing using the same mechanism as DIP. SHiP [33] extends DRRIP by adapting the insertion priority based on the memory address, PC, or instruction sequence. PDP [10] and IRGD [30] use auxiliary monitors to profile the access pattern and periodically recompute their policy.

These two trends show that *(i)* on real access streams, aging reveals information relevant to replacement; and *(ii)* adapting the policy to the access stream improves performance. But these policies do not make the best use of the information they capture, and prior theory does not suggest the right policy.

We use planning theory to design a practical policy that addresses these issues. EVA is intuitive and inexpensive to implement. In contrast to most empirical policies, EVA does not explicitly encode particular heuristics (e.g., preferring recently-used lines). Rather, it is a general approach that aims to make the best use of the limited information available, so that prior heuristics arise naturally when appropriate.

## 3. REPLACEMENT UNDER UNCERTAINTY

All replacement policies have the same goal and face the same constraints. Namely, they try to maximize the cache's hit rate subject to limited cache space. We can precisely characterize these tradeoffs and develop the intuition behind EVA by considering each of them in greater depth.

First, we introduce two random variables, $H$ and $L$. $H$ is the *age*[1] at which the line hits, undefined if it is evicted, and $L$ is the age at which the line's *lifetime*[2] ends, whether by hit or eviction. For example, $\mathrm{P}[H = 8]$ is the probability the line hits at age 8, and $\mathrm{P}[L = 8]$ is the probability that it either hits or is evicted at age 8.

Policies try to maximize the cache's hit rate, which necessarily equals the average line's hit probability:

$$\text{Hit rate} = \mathrm{P}[\text{hit}] = \sum_{a=1}^{\infty} \mathrm{P}[H = a]\,, \tag{1}$$

but are constrained by limited cache space. Since every access starts a new lifetime, the average lifetime equals the cache

[1] *Age* is the number of accesses since the line was last referenced.

[2] We break up time for each line into *lifetimes*, the idle periods between hits or evictions. For example, an address that enters the cache at access 1000, hits at access 1016, and is evicted at access 1064 has two lifetimes of lengths 16 and 48, respectively.

size, $N$ [4, Eq. 1]:

$$N = \sum_{a=1}^{\infty} a \cdot \mathrm{P}[L = a] \tag{2}$$

Comparing these two equations, we see that hits are beneficial irrespective of their age, yet the cost in space increases in proportion to age (the factor of $a$ in Eq. 2). So to maximize the cache's hit rate, the replacement policy must attempt to both maximize hit probability *and* limit how long lines spend in the cache. From these considerations, one can see why MIN is optimal.

*But how should one trade off between these competing, incommensurable objectives under uncertainty?* Obvious generalizations of MIN like evicting the line with the highest expected time until reference or the lowest expected hit probability are inadequate, since they only account for one side of the tradeoff.

**Example:** Inspired by MIN, several empirical policies predict time until reference, and evict the line with the longest predicted time until reference [11, 23, 30, 33]. We present a simple counterexample that shows why predictions of time until reference are inadequate. Although this example may seem obvious to some readers, predicting time until reference is in fact a commonly used replacement strategy.

Suppose the replacement policy has to choose a victim from two candidates: A is referenced immediately with probability $9/10$, and in 100 accesses with probability $1/10$; and B is always referenced in two accesses. In this case, the best choice is to evict B, betting that A will hit immediately, and then evict A if it does not. Doing so yields an expected hit rate of $9/10$, instead of $1/2$ from evicting A. Yet A's expected time until reference is $1 \times 9/10 + 100 \times 1/10 = 10.9$, and B's is 2. Thus, according to their predicted time until reference, A should be evicted. This is wrong.

Predictions fail because they ignore the possibility of future evictions. When behavior is uncertain and changes over time, the replacement policy can learn more about candidates as they age. This means it can afford to gamble that candidates will hit quickly and evict them if they do not. But expected time until reference ignores this insight, and is thus influenced by large reuse distances that will never be reached in the cache. In this example, it is skewed by reuse distance 100. *But the optimal policy never keeps lines this long.* Indeed, the value "100" plays no part in calculating the cache's maximum hit rate, so it is wrong for it to influence replacement decisions. Such situations arise in practice and can be demonstrated on analytical examples (see Appendix A).

# 4. EVA REPLACEMENT POLICY

Since it is inadequate to consider either hit probability or time until reference in isolation, we must reconcile them in a single metric. We resolve this problem by viewing time spent in the cache as forgone hits, i.e. as the opportunity cost of retaining lines. We thus rank candidates by their *economic value added* (EVA), or how many hits the candidate yields over the "average candidate". Sec. 5 shows this intuitive formulation follows from MDP theory, and in fact maximizes the cache's hit rate.

EVA essentially views retaining a line in the cache as an investment, with the goal of retaining the candidates that yield the highest profit (measured in hits). Since cache space is a scarce resource, we need to account for how much space each candidate consumes. We do so by "charging" each candidate for the time it will spend in the cache (its remaining lifetime). We charge candidates at a rate of a line's average hit rate (i.e., the cache's hit rate divided by its size), since this is the long-run opportunity cost of consuming cache space. Hence, the EVA for a candidate of age $a$ is:

$$\mathrm{EVA}(a) = \mathrm{P}\big[\text{hit}|\text{age } a\big] - \frac{\text{Hit rate}}{N} \times \mathrm{E}\big[L - a|\text{age } a\big] \tag{3}$$

**Example:** To see how EVA works, suppose that a candidate has a 20% chance of hitting in 10 accesses, 30% chance of hitting in 20 accesses, and a 50% chance of being evicted in 32 accesses. We would expect to get 0.5 hits from this candidate, but these come at the cost of the candidate spending an expected 24 accesses in the cache. If the cache's hit rate were 40% and it had 16 lines, then a line would yield 0.025 hits per access on average, so this candidate would cost an expected $24 \times 0.025 = 0.6$ forgone hits. Altogether, the candidate yields an expected net $0.5 - 0.6 = -0.1$ hits—its *value added* over the average candidate is negative! In other words, retaining this candidate would tend to *lower* the cache's hit rate, even though its chance of hitting (50%) is larger than the cache's hit rate (40%). It simply takes space for too much time to be worth the investment.

Now suppose that the candidate was not evicted (maybe there was an even less attractive candidate), and it appeared again $t = 16$ accesses later. Since time has passed and it did not hit at $t = 10$, its expected behavior changes. It now has a $30\%/(100\% - 20\%) = 37.5\%$ chance of hitting in $20 - 16 = 4$ accesses, and a $50\%/(100\% - 20\%) = 62.5\%$ chance of being evicted in $32 - 16 = 16$ accesses. Hence we expect 0.375 hits, a lifetime of 13.5 accesses, and forgone hits of $13.5 \times 0.025 = 0.3375$. In total, the candidate yields net 0.0375 hits over the average candidate—after not hitting at $t = 10$, it has become more valuable!

This example shows that EVA can change over time, sometimes in unexpected ways. Our contribution is identifying hit probability and expected lifetime as the two fundamental tradeoffs, and reconciling them within a single metric, EVA. Sec. 5 shows that EVA is indeed the right metric.

We have limited our consideration to the candidate's current lifetime in these examples. But conceptually there is no reason for this, and we should also consider future lifetimes. For simplicity, we assume for now that lines behave the same following a reference. It follows that a candidate's EVA (its difference from the average candidate) is zero after its current lifetime. In Sec. 4.2 we show how EVA can account for future behavior at modest additional complexity by dividing lines into *classes* with different expected behavior, and Sec. 5 formalizes these notions.

## 4.1 Computing EVA

We can generalize these examples using conditional probability and the random variables $H$ and $L$ introduced above. (Sec. 6 discusses how we sample these distributions.) To compute EVA, we compute the candidate's expected current lifetime and its probability of hitting in that lifetime.

Following conventions in MDP theory, we denote EVA at

age $a$ as $h(a)$ and the cache's hit rate as $\overline{g}$. Let $r(a)$ be the expected number of hits, or reward, and $c(a)$ the forgone hits, or cost. Then from Eq. 3:

$$\text{EVA}(a) = h(a) = r(a) - c(a) \tag{4}$$

The reward $r(a)$ is the expected number of hits for a line of age $a$. This is simple conditional probability; age $a$ restricts the sample space to lifetimes at least $a$ accesses long:

$$r(a) = \frac{\text{P}[H > a]}{\text{P}[L > a]} \tag{5}$$

The forgone hits $c(a)$ are $\overline{g}/N$ times the expected lifetime:

$$c(a) = \frac{\overline{g}}{N} \times \frac{\sum_{x=1}^{\infty} x \cdot \text{P}[L = a+x]}{\text{P}[L > a]} \tag{6}$$

To summarize, we select a victim by comparing each candidate's EVA (Eq. 4) and evict the candidate with the lowest EVA. Note that our implementation does not evaluate Eq. 4 during replacement, and instead precomputes ranks for each age and updates them infrequently. Moreover, Eqs. 5 and 6 at age $a$ can each be computed incrementally from age $a+1$, so EVA requires just a few arithmetic operations per age (Sec. 6).

## 4.2 EVA with classification

We now extend EVA to support distinct classes of references, allowing it to distinguish future behavior between candidates. Specifically, we discuss in detail how to extend EVA to support reused/non-reused classification. That is, we divide the cache into two classes: lines that have hit at least once, and newly inserted lines that have not. This simple scheme has proven effective in prior work [11, 16], but unlike prior work EVA does not assume reused lines are preferable to non-reused. Instead, EVA adapts its policy based on observed behavior, computing the EVA for lines of each class while taking into account their expected future behavior.

We denote the EVA of reused lines as $h^R(a)$, and the EVA of non-reused lines as $h^{NR}(a)$. We refer to class $C$ whenever either $R$ or $NR$ apply. With classification, the terms of EVA are further conditioned upon a line's class. For example, the reward for a reused line is:

$$r^R(a) = \frac{\text{P}\big[H > a | \text{reused}\big]}{\text{P}\big[L > a | \text{reused}\big]} \tag{7}$$

Forgone hits and non-reused lines are similarly conditioned.

Without classification, we were able to consider only the current lifetime, assuming all lines behaved identically following a reference. When distinguishing among classes, this is untenable, and we must consider all future lifetimes because each class may differ from the average. The EVA for a single lifetime of class $C$ is unchanged:

$$h^C_\ell(a) = r^C(a) - c^C(a), \tag{8}$$

but the future lifetimes are now important. Specifically, if a lifetime ends in a hit, then the next will be a reused lifetime. Otherwise, if it ends in an eviction, then the next will be non-reused. Thus if the miss rate of class $C$ at age $a$ is $m_C(a)$, then the EVA (extending to infinity) is:

$$h^C(a) = \overbrace{h^C_\ell(a)}^{\text{Current lifetime}} + \overbrace{(1 - m_C(a)) \cdot h^R(0)}^{\text{Hits} \rightarrow \text{Reused}} + \overbrace{m_C(a) \cdot h^{NR}(0)}^{\text{Misses} \rightarrow \text{Non-reused}} \tag{9}$$

Moreover, the average line's EVA is zero by definition, and reused lines are simply those that hit. So if the cache's miss rate is $m$, then:

$$0 = (1-m) \cdot h^R(0) + m \cdot h^{NR}(0) \tag{10}$$

From Eq. 9 and Eq. 10, it follows that:

$$h^R(a) = h^R_\ell(a) + \frac{m - m_R(a)}{m_R(0)} \cdot h^R_\ell(0) \tag{11}$$

$$h^{NR}(a) = h^{NR}_\ell(a) + \frac{m - m_{NR}(a)}{m_R(0)} \cdot h^R_\ell(0) \tag{12}$$

These equations have simplified terms, so their interpretation is not obvious. Essentially, these equations compare the rate that a class is creating additional reused lines ($m - m_C(a)$) to the rate at which lines are leaving the reused class ($m_R(0)$), and their ratio is how similar the class is to reused lines.

**Summary:** Classification ultimately amounts to adding a constant to the unclassified, per-lifetime EVA. If reused and non-reused lines have the same miss rate, then the added terms cancel and EVA reverts to Eq. 4. EVA thus incorporates classification without a fixed preference for either class, instead adapting its policy based on observed behavior.

Although we have focused on reused/non-reused classification, these ideas apply to other classification schemes as well, so long as one is able to express how lines transition between classes (e.g., Eq. 9).

# 5. CACHE REPLACEMENT AS AN MDP

We now formalize cache replacement as a Markov decision process (MDP). We use the MDP to show that *(i)* EVA maximizes hit rate and, using a simple memory reference model, that *(ii)* our implementation follows from a relaxation of the MDP. Finally, we discuss some nice properties that follow from this formulation, which allow us to build a simple and efficient implementation.

## 5.1 Background on Markov decision processes

MDPs [24] are a popular framework to model decision-making under uncertainty, and can yield optimal policies for a wide variety of settings and optimization criteria.

As the name suggests, MDPs extend Markov chains to model decision-making. An MDP consists of a set of states $S$. In state $s \in S$, an action $\alpha$ is taken from the set $A_s$, after which the system transitions to state $s'$ with probability $\text{P}(s'|s,\alpha)$. Finally, each action gives a reward $r(s,\alpha)$.

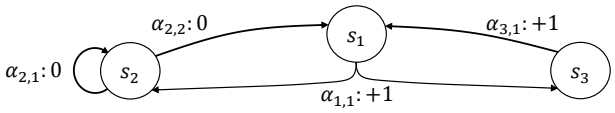



**Figure 1: Example MDP with three states. Actions transition between states probabilistically (e.g., $\alpha_{1,1}$ to $s_2$ or $s_3$) and produce a reward (adjacent to edge label). A state may allow multiple actions (e.g., $s_2$).**

Fig. 1 illustrates the key elements of MDPs with a simple example. This MDP has three states ($s_i$) and four actions (action $\alpha_{i,j}$ denotes the $j^{\text{th}}$ action in state $s_i$). Actions trigger state transitions, which may be probabilistic and have an

associated reward. In this example, all actions except $\alpha_{1,1}$ are deterministic, but $\alpha_{1,1}$ can cause a transition to either $s_2$ or $s_3$. Actions $\alpha_{1,1}$ and $\alpha_{3,1}$ give a reward, while $\alpha_{2,1}$ and $\alpha_{2,2}$ do not. To maximize rewards in the long run, the optimal policy in this case is to take actions $\alpha_{1,1}$, $\alpha_{2,2}$, and $\alpha_{3,1}$ in states $s_1$, $s_2$, and $s_3$, respectively.

The goal of MDP theory is to find the best actions to take in each state to achieve a given objective, e.g. to maximize the total reward. After describing the cache replacement MDP, we give the relevant background in MDP theory needed to describe the optimal cache replacement policy.

## 5.2 Generalized cache replacement MDP

Fig. 2 shows a highly general MDP for cache replacement. The states $s$ consist of some information about candidates, which will vary depending on the memory reference model. For example, in the IRM the state is the reference probability of each candidate, and in MIN it is the time until reference of each candidate.

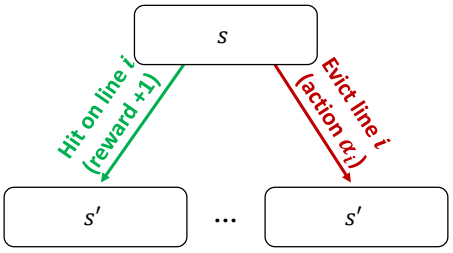



**Figure 2: Generalized cache replacement MDP. State $s$ consists of some information about candidates, and transitions to state $s'$ upon a cache access. Hits give reward 1; evictions let the replacement policy choose a victim.**

Upon every cache access, the state transitions to the state $s'$ (which may be the same as the original state $s$). Misses yield no reward, but the replacement policy now must choose a victim to replace. The replacement policy can take one of $N$ actions $\alpha_1 \ldots \alpha_N$, where $\alpha_i$ means evicting the $i^{\text{th}}$ candidate. Hits yield a reward of 1, and are independent of replacement action. (The replacement policy is not invoked for hits.) Thus the reward is the probability of having hit on any line:

$$r(s,\alpha_j) = \sum_{i=1}^{W} \mathrm{P}\big[\text{hit } i|s\big] \tag{13}$$

To maximize the cache's hit rate, we should optimize the *long-run average reward*.

Given this general model of cache replacement, MDP theory describes the optimal replacement policy. In Sec. 5.4, we will specialize this MDP on a simple memory reference model to show how we arrived at EVA.

## 5.3 Background in optimal MDP policies

This section gives a brief overview of MDP theory, narrowly focused on our application of MDPs to cache replacement. For a comprehensive treatment, see Puterman [24].

The optimal policy depends on the optimization criterion, but the general procedure is common across criteria. Each criterion has an associated optimality equation, which assigns a value to each state based on actions taken. For example, the optimality equation for the expected total reward after $T$ actions is $v^T(s)$:

$$v^T(s) = \max_{\alpha \in A_s} \left\{ r(s,\alpha) + \sum_{s' \in S} \mathrm{P}(s'|s,\alpha) v^{T+1}(s') \right\} \tag{14}$$

Eq. 14 says that the maximum total reward is achieved by taking the action that maximizes the immediate reward plus the expected total future reward. For an infinite horizon, Eq. 14 drops the superscripts and becomes recursive on $v$.

The significance of the optimality equation is that any $v^*$ that satisfies the optimality equation gives the maximum value over all policies. Hence, a solution $v^*$ directly yields an optimal policy: take the action chosen by the max operator in Eq. 14. Optimality equations therefore summarize all relevant information about present and future rewards.

**Long-run average reward:** The long-run average reward can be intuitively described using the expected total reward. If the expected total reward after $T$ actions is $v^T(s)$, then the expected average reward, or *gain*, is $g^T(s) = v^T(s)/T$. The long-run average reward is just the limit as $T$ goes to infinity. However, since all states converge to the same average reward in the limit, the gain is constant $g(s) = \overline{g}$ and does not suffice to construct an optimal policy. That is, the cache converges to the same hit rate from every initial state, so the long-run hit rate cannot by itself distinguish among states.

To address this, MDP theory introduces the *bias* of each state $h(s)$, which represents how much the total reward from state $s$ differs from the mean. In other words, after $T$ accesses one would expect a total reward (i.e., cumulative hits) of $T\,\overline{g}$; the bias is the asymptotic difference between the expected total reward from $s$ and this value:

$$h(s) = \lim_{T \to \infty} v^T(s) - T\,\overline{g} \tag{15}$$

In the limit, the bias is a finite but generally non-zero number. The bias is also called the *transient reward*, since in the long run, it becomes insignificant relative to the total reward $v^T(s)$ once all states have converged to the average reward $\overline{g}$.

Perhaps surprisingly, this transient reward indicates how to achieve the best long-run average reward. Optimizing the bias is equivalent to optimizing the expected total reward after $T$ actions, and choosing $T$ large enough effectively optimizes the long-run average reward (to within arbitrary precision). For rigorous proofs, see [24, §8]. The optimality equation for the long-run average reward is:

$$\overline{g} + h(s) = \max_{\alpha \in A_s} \left\{ r(s,\alpha) + \sum_{s' \in S} \mathrm{P}(s'|s,\alpha) h(s') \right\} \tag{16}$$

Since solving Eq. 16 yields an optimal policy, the optimal policy for long-run average reward MDPs is to *take the action that maximizes the immediate reward plus expected bias.*

## 5.4 An MDP for dynamic behavior

We now show how to tractably model dynamically changing reference probabilities by specializing the MDP in Fig. 2. We do so using a simple memory reference model, called the *iid reuse distance model*. Rather than assuming cache lines behave non-uniformly with static reference probabilities, this model asserts they behave largely *homogeneously* and share *dynamic* reference probabilities.

**Memory reference model:** The iid reuse distance model divides references into a small number of classes, and models that reuse distances[3] are independent and identically distributed within each class. That is, we model that each reference belongs to some class $c$, with reuse distance $d$ distributed according to $\mathrm{P}[D_c = d]$.

This model is simple to analyze and captures the most important features of dynamic program behavior: changing reference probabilities, and time spent in the cache (Sec. 3). It is reasonably accurate because the private cache levels filter out successive references to the same line, so the LLC naturally sees an access stream stripped of short-term temporal locality [5, 14, 16]. The iid reuse distance model is not a perfect description of programs, but the reference patterns it models poorly are uncommon at the LLC. The iid model is therefore a much better model of the LLC reference streams than prior models. What matters is that the model is simple enough that one can find the statistically optimal policy, and accurate enough that it captures the important tradeoffs so that this policy is competitive.

Ideally, we would validate these assumptions through a statistical test against real applications. Unfortunately, there is no robust statistical test for whether a dataset is iid that does not itself depend on other assumptions on how the data were generated. Instead, we justify the model by applying it to problems. Prior work has used the iid model to predict cache performance [4], and both PDP [10] and IRGD [30] use a similar model (albeit informally, without studying the optimal policy). Other work makes similar assumptions for other purposes [5, 10, 25, 26]. Appendix B shows the error from assuming independence is small. And our results show that the model yields a statistically optimal policy that outperforms state-of-the-art empirical policies.

**State space:** Every line in the set has reuse distance that is iid, letting us infer its time until reference. The probability that a line of age $a$ and class $c$ has time until reference $t$ is the conditional probability:

$$\mathrm{P}\big[D_c = t + a | \text{age } a\big] = \frac{\mathrm{P}[D_c = t + a]}{\mathrm{P}[D_c \geq a]} \tag{17}$$

With $N$ lines, an MDP state $s$ is a tuple:

$$s = (c_1, a_1, c_2, a_2 \ldots c_N, a_N), \tag{18}$$

where each $a_i/c_i$ is the age/class of the $i^{\text{th}}$ line. This state space encodes the information directly observable to the replacement policy. Moreover, it is sufficient within our model to encode *all* available information about each candidate.[4]

**State transitions:** Every access to the cache causes a state transition. There are two possible outcomes to every cache access (Fig. 3):

1. *Hits:* Each line hits with probability that its time until reference is zero (Eq. 17). Each line that hits then transitions to age 1 and may transition to a new class (i.e., become reused). Lines that do not hit age by one, i.e. $a_i$ ages to $a_i' = a_i + 1$.

[3] A reference's *reuse distance* is the number of accesses between references to the same address; contrast with *stack distance*, the number of unique addresses referenced.

[4] Our MDP thus encodes the *belief-states* of a partially observable MDP [20] using lines' time until reference.

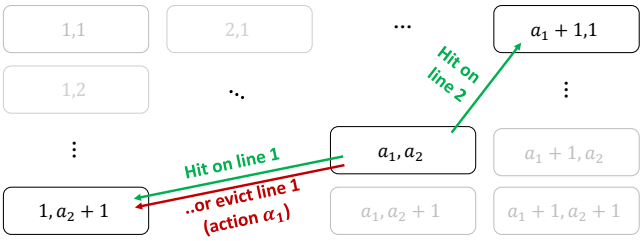



**Figure 3: Cache replacement MDP for a 2-line cache with a single reference class. The state $s = (a_1, a_2)$ is lines' ages; each hit gives reward 1; a line's age resets to 1 upon hit or eviction.**

2. *Evictions:* If no line hits, then the replacement policy must choose a line to evict. To evict line $j$, it takes action $\alpha_j$. In this case, line $j$ transitions to age $a_j' = 1$ and the incoming reference's class (i.e., non-reused), and all other lines age by one as before.

It is unnecessary to explicitly compute the state transition probabilities to describe the optimal policy. Although the optimal policy can be solved for explicitly, doing so at runtime is impractical. (Given a maximum reuse distance of $d_{\max}$, this MDP has ${d_{\max}}^W$ states. Even after compressing the state space, it has intractably many states.) The MDP formulation is useful for its insight, but not as a practical policy itself. Instead, we show that EVA from Sec. 4 approximates the optimal MDP policy.

**A practical implementation of the MDP:** Since the immediate reward (Eq. 13) is independent of action, the optimal replacement policy is to maximize the expected future bias (Eq. 16). Bias is the expected reward minus the expected average reward over the same number of actions—*bias is exactly EVA* (Sec. 4). Moreover, MDP theory guarantees that this approach maximizes the hit rate in the long run.

The main approximation in EVA is to decompose the policy from *MDP states* to *individual candidates*. That is, rather than selecting a victim by looking at all ages simultaneously, our implementation ranks candidates individually and evicts the one with highest rank. This approach is common in practical policies [10, 11, 30, 33], and we believe it introduces little error to the MDP solution because hits occur to individual lines, not the entire set.

Informally, the idea is that *(i)* the cache's EVA is simply the sum of every line's EVA; *(ii)* the cache's EVA is thus maximized by retaining the lines with the highest EVA; and *(iii)* following MDP theory, this maximizes the cache's hit rate. We thus compute EVA for each candidate, conditioning upon its age and class as suggested by the iid model.

**Related work:** Prior work has used MDPs to study replacement within the IRM [1, 3]. Our MDP is similar, but with the critical difference that we tractably model dynamic behavior. This is essential to achieve good performance. Prior approaches are too complex: Aho et al. [1] propose modeling the reference history to capture dynamic behavior, but deem this approach intractable and do not consider it in depth.

Since the IRM assumes static, heterogeneous reference probabilities, their MDP states are the *addresses* cached. In contrast, we use the iid reuse distance model, so we ignore addresses. Instead, the MDP states are the *ages* and *classes*

cached. This lets us tractably capture dynamic behavior and produce a high-performance policy.

## 5.5 Generalized optimal replacement

MDPs are capable of modeling a wide variety of memory reference models and cache replacement problems. We have presented an MDP for the iid reuse distance model, but it is easy to extend MDPs to other contexts. For example, prior work models the IRM with uniform [1] and non-uniform [3] replacement costs. With small modifications, our MDP can model a compressed cache [21], where candidates have different sizes and conventional prediction-based policies are clearly inadequate (Appendix C). The above discussion applies equally to such problems, and shows that EVA is a general principle of optimal replacement under uncertainty.

In particular, EVA subsumes prior optimal policies, i.e. MIN and $A_0$ (the IRM-optimal policy), in their respective memory reference models. MIN operates on lines' times until reference. Since all hits yield the same reward, EVA is maximized by minimizing forgone hits, i.e. by minimizing time until reference. In the IRM, $A_0$ operates on lines' reference probabilities. Reward is proportional to hit probability and forgone hits are inversely proportional to hit probability, so EVA is maximized by evicting lines with the lowest hit probability. EVA thus yields the optimal policy in both reference models.

The real contribution of the MDP framework, however, lies in going beyond prior reference models. Unlike predicting time until reference, the MDP approach does not ignore future evictions. It therefore gives an accurate accounting of all outcomes. In other words, EVA generalizes MIN.

## 5.6 Qualitative gains from planning theory

**Convergence:** MDPs are solved using iterative algorithms that are guaranteed to converge to an optimal policy within arbitrary precision. One such algorithm is *policy iteration* [17], wherein the algorithm alternates between computing the value $v^T$ for each state and updating the policy. Our implementation uses an analogous procedure, alternatively monitoring the cache's behavior (i.e., "computing $v^T$") and updating ranks. Although this analogy is no strict guarantee of convergence, it gives reason to believe that our implementation converges on stable applications, as we have observed empirically.

**Convexity:** Beckmann and Sanchez [5] proved under similar assumptions to the iid model that MIN is *convex*; i.e., it yields diminishing gains in hit rate with additional cache space. Convexity is important to simplify cache management. Their proof does not rely on any specific properties of MIN, it only requires that *(i)* the policy is optimal in some model and *(ii)* its miss rate can be computed for any arbitrary cache size. Since EVA is iid-optimal and the MDP yields the miss rate, the proof holds for EVA as well. Thus EVA is convex, provided that the iid model and other assumptions hold. Empirically, we observe convex performance with an idealized implementation, and minor non-convexities with coarsened ages (Sec. 6.1).

# 6. IMPLEMENTATION

One contribution of the MDP formulation is that it suggests a simple implementation of EVA, shown in Fig. 4: *(i)* A small table, called the *eviction priority array*, ranks candidates to

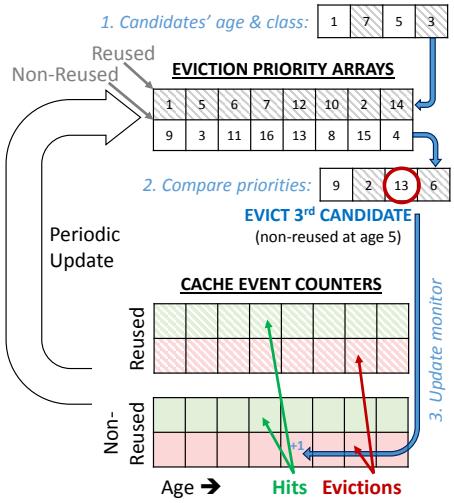



**Figure 4: An efficient implementation of EVA.**

select a victim. *(ii)* Counters record the age distribution of hits of evictions. And infrequently *(iii)* a lightweight software runtime computes EVA from these counters, and then updates the eviction priorities.

This implementation requires trivial hardware: narrow comparisons and increments plus a small amount of state. All of the analytical complexity is pushed to software, where updates add small overheads. This hybrid design is broadly similar to prior work in cache partitioning [5]; however, we also describe a hardware-only implementation in Sec. 6.4.

Unlike prior policies, EVA does not devote a fraction of sets to monitoring alternative policies (cf. [11, 26]), nor does it require auxiliary tags to monitor properties independent of the replacement policy (cf. [5, 10, 30]). This is possible because of EVA's analytical foundation: it converges to the right policy simply by monitoring its own behavior (Sec. 5.6, cf. [33]). As a result, our implementation makes full use of the entire cache and eliminates overheads from monitors.

## 6.1 Hardware operations

**Aging:** We use per-set, coarsened ages [10]. Each cache line has a $k$-bit age, and each set has a $j$-bit counter that is incremented upon an access to the set ($k = 7$ and $j = 4$ in our evaluation). When a set's counter reaches a value $A$, it is reset and every age in the set is incremented until saturating at $2^k - 1$. Software adapts $A$ to the access pattern (Sec. 6.2).

**Ranking:** To rank candidates cheaply, we use a small *eviction priority array*. We use each candidate's age to index into the priority array, and evict the candidate with the highest eviction priority. We set priorities such that if age $a$ is ranked higher than age $b$, then $h(a) < h(b)$ (Sec. 6.2). To ensure that lines are eventually evicted, saturated ages always have the highest eviction priority.

To work with classification, we add a reused bit to each cache tag, and use two priority arrays to store the priorities of reused and non-reused lines. Eviction priorities require $2^{k+1} \times (k+1)$ bits, or 256 B with $k = 7$.

The eviction priority array is dual ported to support peak memory bandwidth. With 16 ways, we can sustain one eviction every 8 cycles, for 19.2 GBps per LLC bank.

**Event counters:** To sample the hit and lifetime distributions,

we add two arrays of 16-bit counters ($2^k \times 16\,\text{b} = 256\,\text{B}$ per array) that record the age histograms of hits and evictions. When a line hits or is evicted, the cache controller increments the counter in the corresponding array. These counters are periodically read to update the eviction priorities. To support classification, there are two arrays for both reused and non-reused lines, or 1 KB total with $k = 7$.

## 6.2 Software updates

Periodically (every 256 K accesses), an OS runtime adapts EVA to running applications by setting the aging granularity and eviction priorities. First, we obtain the hit and lifetime distributions ($\text{P}[H = a]$ and $\text{P}[L = a]$) from the counters, and average them with prior values to maintain a history of application behavior. (We use an exponential moving average with coefficient 0.8.)

**Adaptive aging:** To make a fair comparison with prior policies, we evaluate EVA using minimal tag state (Fig. 9). This necessitates adapting the age granularity $A$ because, with very few age bits (<5 b), no single value for $A$ works well across all applications. The challenge is to set the granularity large enough for the cache to observe relevant behavior, but small enough that it can distinguish between behaviors at different ages. We achieve this through the following heuristic: If the cache has no hits *or* if more than 10% of hits and evictions occur at the maximum age, then increase $A$ by one. Otherwise, if less than 10% of hits occur in the second half of ages (ages $\geq 2^{k-1}$), then decrease $A$ by one. We find that this simple heuristic rapidly converges to the right age granularity across all our evaluated applications.

Adaptive aging is a practical compromise to perform well with minimal state. However, with larger tags (e.g., with 8 b tags, as in most of our evaluation), adaptive aging can be disabled with no performance loss.

**Updating ranks:** We compute EVA in a small number of arithmetic operations per age, and sort the result to find the eviction priority for each age and class.

Updates occur in four passes over ages, shown in Algorithm 1. In the first pass, we compute $m_R$, $m_{NR}$, and $m = 1 - \overline{g}$ by summing counters. Second, we compute lifetime EVA incrementally: Eq. 8 requires five additions, one multiplication, and one division per age. Third, classification (e.g., Eq. 9) adds one more addition and multiplication per age. Finally, we sort EVA to find the final eviction priorities. Our C++ implementation takes just 123 lines of code and incurs negligible runtime overheads (see below).

**Algorithm 1.** Algorithm to compute EVA and update ranks.

**Inputs:** hitCtrs, evictionCtrs — event counters, $A$ — age granularity
**Returns:** rank — eviction priorities for all ages and classes

```
1:  function UPDATE
2:      for a ← 2^k to 1:                          ▷ Miss rates from summing over counters.
3:          for c ∈ {nonReused, reused}:
4:              hits_c += hitCtrs[c, a]
5:              misses_c += evictionCtrs[c, a]
6:          m_R[a] ← misses_R / (hits_R + misses_R)
7:          m_NR[a] ← misses_NR / (hits_NR + misses_NR)
8:      m ← (hits_R + hits_NR) / (misses_R + misses_NR)
9:      perAccessCost ← (1 − m) × A/S
10:     for c ∈ {nonReused, reused}:               ▷ Compute EVA backwards over ages.
11:         expLifetime, hits, events ← 0
12:         for a ← 2^k to 1:
13:             expectedLifetime += events
14:             eva[c, a] ← (hits − perAccessCost × expectedLifetime) / events
15:             hits += hitCtrs[c, a]
16:             events += hitCtrs[c, a] + evictionCtrs[c, a]
17:     evaReused ← eva[reused, 1] / m_R[0]          ▷ Differentiate classes.
18:     for c ∈ {nonReused, reused}:
19:         for a ← 2^k to 1:
20:             eva[c, a] += (m − m_c[a]) × evaReused
21:     order ← ARGSORT(eva)                        ▷ Finally, rank ages by EVA.
22:     for i ← 1 to 2^(k+1):
23:         rank[order[i]] ← 2^(k+1) − i
24:     return rank
```

| | Area | | Energy | |
|---|---|---|---|---|
| | ($\text{mm}^2$) | (% 1 MB LLC) | (nJ / LLC miss) | (% 1 MB LLC) |
| **Ranking** | 0.010 | 0.05% | 0.014 | 0.6% |
| **Counters** | 0.025 | 0.14% | 0.010 | 0.4% |
| **8-bit Tags** | 0.189 | 1.07% | 0.012 | 0.5% |
| **H/W Updates (Optional, Sec. 6.4)** | 0.052 | 0.30% | 380 / 128 K | 0.1% |

**Table 1: Implementation overheads at 65 nm.**

## 6.3 Overheads

**Ranking and counters:** Our implementation adds 1 KB for counters and 256 B for priority arrays. We have synthesized our design in a commercial process at 65 nm at 2 GHz. We lack access to an SRAM compiler, so we use CACTI 5.3 [31] for all SRAMs (using register files instead of SRAM for all memories makes the circuit 4× larger). Table 1 shows the area and energy for each component at 65 nm. Absolute numbers should be scaled to reflect more recent technology nodes. We compute overhead relative to a 1 MB LLC using area from a 65 nm Intel E6750 [9] and energy from CACTI. Overheads are small, totaling 0.2% area and 1.0% energy over a 1 MB LLC. Total leakage power is 2 mW. Even with one LLC access every 10 cycles, EVA adds just 7 mW at 65 nm, or 0.01% of the E6750's 65 W TDP [9].

**Software updates:** Updates complete in a few tens of K cycle. Specifically, with $k = 7$ age bits, updates take 43 K cycle on an Intel Xeon E5-2670, and at small $k$ scale roughly in proportion to the maximum age ($2^k$). Because updates are infrequent, the runtime and energy overhead is negligible: conservatively assuming that updates are on the critical path, we observe that updates take an average 0.1% of system cycles and a maximum of 0.3% on SPEC CPU2006 apps.

**Tags:** Since the new components introduced by EVA add negligible overheads, the main overhead is additional tag state. Our implementation uses 8 bits per tag (vs. 2 bits for SHiP). This is roughly 1% area overhead, 0.5% energy overhead, and 20 mW leakage power. However, our evaluation shows that EVA is competitive with prior policies when using fewer age bits. So why do we use larger tags? *We use larger tags because doing so produces a more area-efficient design.* Unlike prior policies, EVA's performance steadily improves with more tag bits (Fig. 9). EVA trades off larger tags for improved performance, making better use of the 99% of cache area not devoted to replacement, and therefore saves area at iso-performance.

**Complexity:** A common concern with analytical techniques like EVA is their perceived complexity. However, we should be careful to distinguish between conceptual complexity—equations, MDPs, etc.—and implementation complexity. In hardware, EVA adds only narrow increments and comparisons; in software, EVA adds a short, low-overhead reconfiguration procedure (Algorithm 1).

## 6.4 Alternative hardware-only implementation

Performing updates in software instead of hardware provides several benefits. Most importantly, software updates reduce implementation complexity, since EVA's implementation is otherwise trivial. Software updates may also be preferable to integrate EVA with other system objectives (e.g., cache partitioning [32]), or on systems with dedicated OS cores (e.g., the Kalray MPPA-256 [8] or Fujitsu Sparc64 XIfx [34]).

However, if software updates are undesirable, we have also implemented and synthesized a custom microcontroller that performs updates in hardware. In hardware, updates are off the critical path and can thus afford to be simple. Our microcontroller computes EVA using a single adder and a small ROM microprogram. We use fixed-point arithmetic, requiring $2^{k+1} \times 32$ bits to store the results plus seven 32-bit registers, or 1052 B with $k = 7$. We have also implemented a small FSM to compute the eviction priorities, which performs an argsort using the merge sort algorithm, adding $2 \times 2^{k+1} \times (k+1)$ bits, or 512 B.

This microcontroller adds small overheads (Table 1)—1.5 KB of state and simple logic—and is off the critical path. It was fully implemented in Verilog by a non-expert in one week, and its complexity is low compared to microcontrollers shipping in commercial processors (e.g., Intel Turbo Boost [27]). So though we believe software updates are a simpler design, EVA can be implemented entirely in hardware if desired.

# 7. EVALUATION

We now evaluate EVA over diverse benchmark suites and configurations. We show that EVA performs consistently well across benchmarks, and consequently outperforms existing policies, closes the gap with MIN, and saves area at iso-performance.

## 7.1 Methodology

We use zsim [28] to simulate systems with 1 and 8 OOO cores with parameters shown in Table 2. We simulate LLCs with sizes from 1 to 8 MB. The single-core chip runs single-threaded SPEC CPU2006 apps, while the 8-core chip runs multi-threaded apps from SPEC OMP2012. Our results hold across different LLC sizes, benchmarks (e.g., PBBS [29]), and with a strided prefetcher validated against real Westmere systems [28].

**Policies:** We evaluate *how well policies use information* by comparing against random and MIN; these policies represent the extremes of no information and perfect information, respectively. We further compare EVA with LRU, RRIP variants (DRRIP and SHiP), and PDP, which are implemented as proposed. We sweep configurations for each policy and select the one that is most area-efficient at iso-performance. DRRIP uses $M = 2$ bits and $\varepsilon = 1/32$ [11]. SHiP uses $M = 2$ bits and PC signatures with idealized, large history counter tables [33]. DRRIP is only presented in text because it performs similarly to SHiP, but occasionally slightly worse. These policies' performance degrades with larger $M$ (Fig. 9). PDP uses an idealized implementation with large timestamps.

**Area:** Except where clearly noted, our evaluation compares policies' performance against their *total cache area at 65 nm*, including all replacement overheads. For each LLC size, we use CACTI to model data and tag area, using the default tag size of 45 b. We then add replacement tags and other overheads, taken from prior work. When overheads are unclear, we use favorable numbers for other policies: DRRIP and SHiP add 2-bit tags, and SHiP adds 1.875 KB for tables, or 0.1 mm$^2$. PDP adds 3-bit tags and 10 K NAND gates (0.02 mm$^2$). LRU uses 8-bit tags. Random adds no overhead. Since MIN is our upper bound, we also grant it zero overhead. Finally, EVA adds 8-bit tags and 0.04 mm$^2$, as described in Sec. 6.

| | |
|---|---|
| **Cores** | Westmere-like OOO [28] at 4.2 GHz; 1 (ST) or 8 (MT) |
| **L1 caches** | 32 KB, 8-way set-assoc, split D/I, 1-cycle |
| **L2 caches** | Private, 256 KB, 8-way set-assoc, inclusive, 7-cycle |
| **L3 cache** | Shared, 1 MB–8 MB, non-inclusive, 27-cycle; 16-way, hashed set-assoc |
| **Coherence** | MESI, 64 B lines, no silent drops; seq. consistency |
| **Memory** | DDR-1333 MHz, 2 ranks/channel, 1 (ST) or 2 (MT) channels |

**Table 2: Configuration of the simulated systems for single- (ST) and multi-threaded (MT) experiments.**

**Workloads:** We execute SPEC CPU2006 apps for 10 B instructions after fast-forwarding 10 B instructions. Since IPC is not a valid measure of work in multi-threaded workloads [2], we instrument SPEC OMP2012 apps with heartbeats. Each completes a region of interest (ROI) with heartbeats equal to those completed in 1 B cycles with an 8 MB, LRU LLC (excluding initialization).

**Metrics:** We report misses per thousand instructions (MPKI) and end-to-end performance; for multi-threaded apps, we report MPKI by normalizing misses by the instructions executed on an 8 MB, LRU LLC. We report speedup using IPC (single-threaded) and ROI completion time (multi-threaded). Results for EVA include overheads from software updates.

## 7.2 Single-threaded results

Fig. 5 plots MPKI vs. cache area for ten representative, memory-intensive SPEC CPU2006 apps. Each point on each curve represents increasing LLC sizes from 1 to 8 MB. First note that the total cache area at the same LLC size (i.e., points along $x$-axis) is hard to distinguish across policies. This is because replacement overheads are small—less than 2% of total cache area.

In most cases, MIN outperforms all practical policies by a large margin. Excluding MIN, some apps are insensitive to replacement policy. On others, random replacement and LRU perform similarly; e.g., `mcf` and `libquantum`. In fact, random often outperforms LRU.

**EVA performs consistently well:** SHiP and PDP improve performance by correcting LRU's flaws on particular access patterns. Both perform well on `libquantum` (a scanning benchmark), `sphinx3`, and `xalancbmk`. However, their performance varies considerably across apps. For example, SHiP performs particularly well on `perlbench`, `mcf`, and `cactusADM`. PDP performs particularly well on `GemsFDTD` and `lbm`, where SHiP exhibits pathologies and performs similar to random replacement.

EVA matches or outperforms SHiP and PDP on most apps and cache sizes. This is because EVA employs a generally optimal strategy within a model that accurately represents memory references, so the right replacement strategies naturally emerge from EVA where appropriate. As a result, EVA

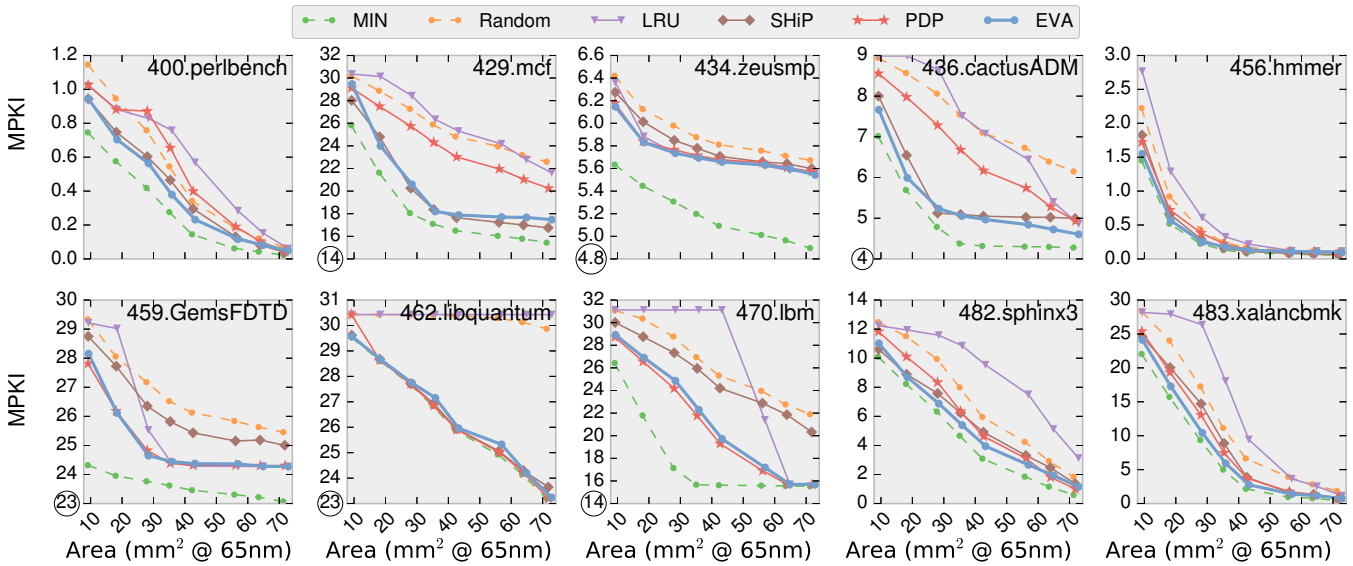



**Figure 5: Misses per thousand instructions (MPKI) vs. total cache area across sizes (1 MB–8 MB) for MIN, random, LRU, SHiP, PDP, and EVA on selected memory-intensive SPEC CPU2006 benchmarks. (Lower is better.)**

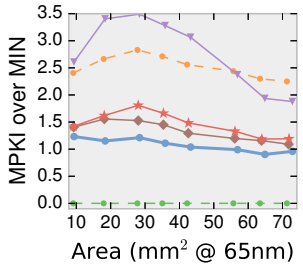



**Figure 6: MPKI above MIN for each policy on SPEC CPU2006.**

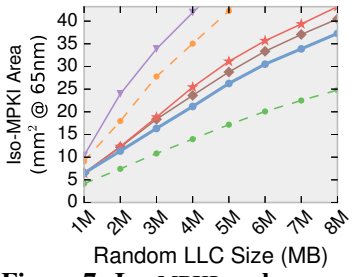



**Figure 7: Iso-MPKI cache area: avg MPKI equal to random at different LLC sizes.**

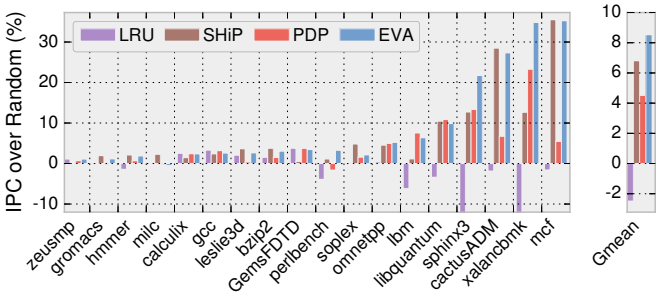



**Figure 8: Performance across SPEC CPU2006 apps on a 35 mm² LLC. (Only apps with >1% difference shown.)**

successfully captures the benefits of SHiP and PDP within a common framework, and sometimes outperforms both. Since EVA performs consistently well, but SHiP and PDP do not, EVA achieves the lowest MPKI of all policies on average.

The cases where EVA performs slightly worse arise for two reasons. First, in some cases (e.g., `mcf` at 1 MB), the access pattern changes significantly between policy updates. EVA can take several updates to adapt to the new pattern, during which performance suffers. But in most cases the access pattern changes slowly, and EVA performs well. Second, our implementation coarsens ages, which can cause small performance variability for some apps (e.g., `libquantum`). Larger timestamps eliminate these variations.

**EVA edges closer to optimal replacement:** Fig. 6 compares the practical policies against MIN, showing the average MPKI gap over MIN across the most memory-intensive SPEC CPU2006 apps[5]—i.e., each policy's MPKI minus MIN's at equal area. One would expect a practical policy to fall somewhere between random replacement (no information) and MIN (perfect information). But LRU actually performs *worse than random* at many sizes because private caches strip out most temporal locality before it reaches the LLC, leaving scanning patterns that are pathological in LRU. In contrast, both SHiP and PDP significantly outperform random replacement. Finally, EVA performs best at equal cache area. On average, EVA closes 57% of the random-MIN MPKI gap. In comparison, DRRIP (not shown) closes 41%, SHiP 47%, PDP 42%, and LRU –9%. *EVA is the first statistically optimal policy to outperform state-of-the-art empirical policies.*

**EVA saves cache space:** Because EVA improves performance, it needs less cache space than other policies to achieve a given level of performance. Fig. 7 shows the *iso-MPKI total cache area* of each policy, i.e. the area required to match random replacement's average MPKI for different LLC sizes (lower is better). For example, a 21.5 mm² EVA cache achieves the same MPKI as a 4 MB cache using random replacement, whereas SHiP needs 23.6 mm² to match this performance.

EVA is the most area efficient over the full range. On average, EVA saves 8% total cache area over SHiP, the best practical alternative. However, note that MIN saves 35% over EVA, so there is still room for improvement.

**EVA achieves the best end-to-end performance:** Fig. 8 shows the IPC speedups over random replacement at 35 mm², the area of a 4 MB LLC with random replacement. Only benchmarks that are sensitive to replacement are shown, i.e. benchmarks whose IPC changes by at least 1% under some policy.

[5]All with ≥3 L2 MPKI; Fig. 5 plus `bzip2`, `gcc`, `milc`, `gromacs`, `leslie3d`, `gobmk`, `soplex`, `calculix`, `omnetpp`, and `astar`.

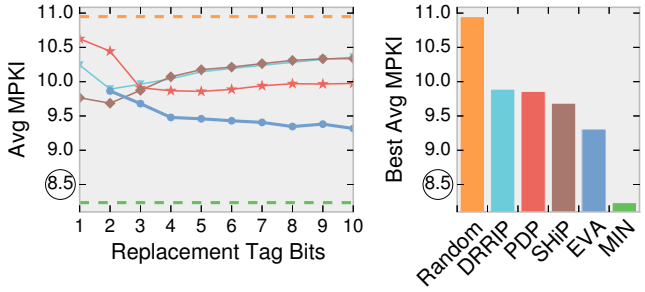



**Figure 9: Avg MPKI for different policies at 4 MB vs. tag overheads (lower is better).**

EVA achieves consistently good speedups across apps, while prior policies do not. SHiP performs poorly on `xalancbmk`, `sphinx3`, and `lbm`, and PDP performs poorly on `mcf` and `cactusADM`. Consequently, EVA achieves the best speedup overall. Gmean speedups on sensitive apps (those shown) are for EVA 8.5%, DRRIP (not shown) 6.7%, SHiP 6.8%, PDP 4.5%, and LRU –2.3%.

**EVA makes good use of additional state:** Fig. 9 sweeps the number of tag bits for different policies and plots their average MPKI at 4 MB. (This experiment is not iso-area.) The figure shows the best configuration on the right; EVA and PDP use idealized, large timestamps. Prior policies achieve peak performance with 2 or 3 bits, after which their performance flattens or even degrades.

With 2 bits, EVA performs better than PDP, similar to DRRIP, and slightly worse than SHiP. Unlike prior policies, EVA's performance improves steadily with more state, and its peak performance exceeds prior policies by a good margin. Comparing the best configurations, *EVA's improvement over SHiP is 1.8× greater than SHiP's improvement over DRRIP*. EVA with 8 b tags performs as well as an idealized implementation, yet still adds small overheads. These overheads more than pay for themselves, saving area at iso-performance (Fig. 7).

**EVA outperforms predictions of time until reference:** We have also evaluated an analytical policy that monitors the distribution of reuse distances and computes candidates' time until reference (Eq. 17). We use large, idealized monitors, and references with immeasurably large reuse distances use twice the maximum measurable distance.

This policy performs poorly compared to EVA and other high-performance policies (graphs omitted due to space constraints). In fact, it often performs worse than random replacement, since predictions of time until reference lead to pathologies like those discussed in Sec. 3. It closes just 29% of the random-MIN MPKI gap, and its gmean IPC speedup vs. random is just 1.4%. It handles simple, regular access patterns well (e.g., `libquantum`), but fails on more complex patterns. These results support the claim that EVA is the correct generalization of MIN under uncertainty.

## 7.3 Multi-threaded results

Fig. 10 extends our evaluation to multi-threaded apps from SPEC OMP2012. Working set sizes vary considerably, so we consider LLCs from 1 to 32 MB, with area shown in log scale on the *x*-axis. All qualitative claims from single-threaded apps hold for multi-threaded apps. Many apps are streaming or access the LLC infrequently; we discuss four representative

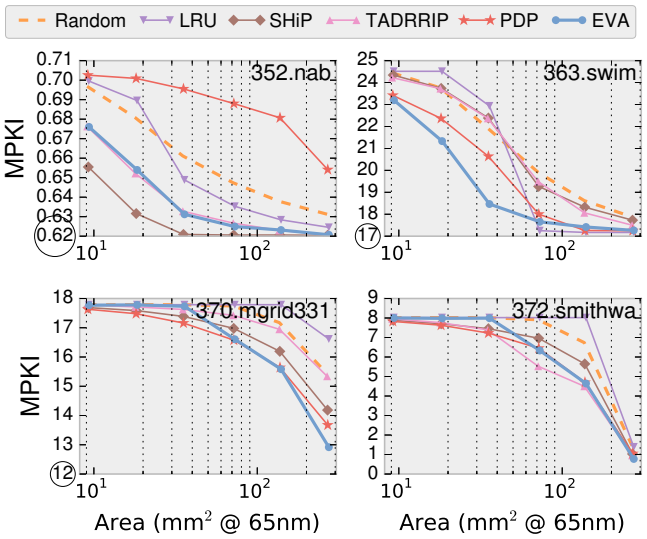



**Figure 10: Misses per thousand instructions (MPKI) vs. total cache area across sizes (1 MB, 2 MB, 4 MB, 8 MB, 16 MB, and 32 MB) for random, LRU, SHiP, PDP, and EVA on selected SPEC OMP2012 benchmarks. (Lower is better.)**

benchmarks from the remainder.

As with single-threaded apps, SHiP and PDP improve performance on different apps. SHiP outperforms PDP on some apps (e.g., `nab`), and both perform well on others (e.g., `smithwa`). Unlike in single-threaded apps, however, DRRIP and thread-aware DRRIP (TADRRIP) outperform SHiP. This difference is largely due to a single benchmark: `smithwa` at 8 MB.

EVA performs well in nearly all cases and achieves the highest speedup. On the 7 OMP2012 apps that are sensitive to replacement,[6] the gmean speedup over random for EVA is 4.5%, DRRIP (not shown) 2.7%, TA-DRRIP 2.9%, SHiP 2.3%, PDP 2.5%, and LRU 0.8%.

# 8. CASE STUDY: GENERALIZING EVA ON COMPRESSED CACHES

To demonstrate that EVA generalizes to new contexts, we evaluate EVA on compressed caches where candidates have different sizes. We compare C-EVA (EVA extended to compressed caches, Appendix C) against CAMP [21], the state-of-the-art policy for compressed caches. C-EVA computes opportunity cost per *byte*, not per cache block. CAMP extends RRIP by adapting the insertion priority for different line sizes and ranks candidates by dividing their RRIP priority by their size. The cache uses BDI compression [22] and an idealized tag directory that only evicts lines when a set runs out of space in the data array.

Similar to above, Fig. 11 evaluates benchmarks that show at least 1% performance improvement for some policy. Compared to random replacement, C-EVA improves performance by 7.9%, CAMP by 6.2%, DRRIP by 5.1%, and LRU by -1.9%. *C-EVA's improvement over CAMP is 1.5× greater than CAMP's improvement over DRRIP*. Thus, EVA generalizes to new contexts and still outperforms the state-of-the-art. Once again, these results support the claim that EVA is the correct general-

[6] Fig. 10 plus `md`, `botsspar`, and `kd`.

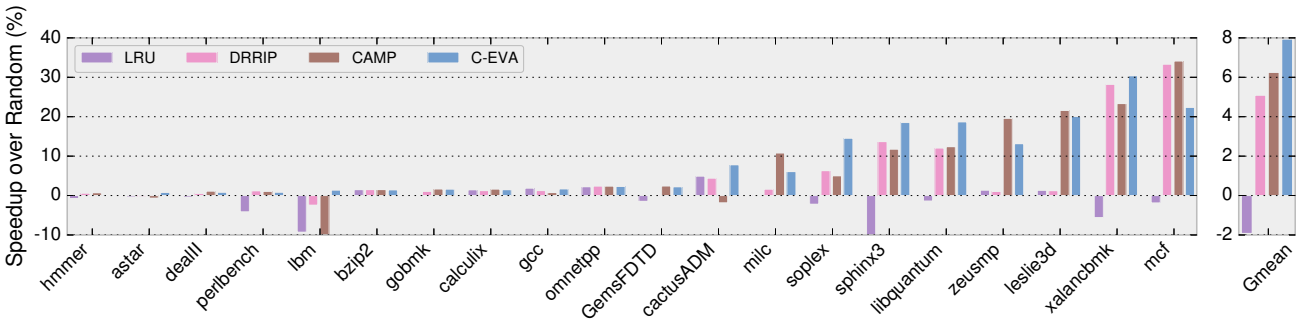



**Figure 11: Performance across SPEC CPU2006 apps on a 4 MB LLC with BDI compression [22]. Only apps with >1% performance difference shown.**

ization of MIN under uncertainty.

## 9. CONCLUSION

We have argued for cache replacement by *economic value added* (EVA). We showed that predicting time until reference, a common replacement strategy, is sub-optimal, and we used first principles of caching and MDP theory to motivate EVA as an alternative principle. We developed a practical implementation of EVA that outperforms existing high-performance policies nearly uniformly on single- and multi-threaded benchmarks. EVA thus gives a theoretical grounding for practical policies, bridging the gap between theory and practice.

## 10. ACKNOWLEDGEMENTS

We thank David Karger for pointing us towards Markov decision processes. This work was supported in part by NSF grant CCF-1318384.

## 11. REFERENCES

[1] A. Aho, P. Denning, and J. Ullman, "Principles of optimal page replacement," *J. ACM*, 1971.

[2] A. Alameldeen and D. Wood, "Ipc considered harmful for multiprocessor workloads," *IEEE Micro*, vol. 26, no. 4, 2006.

[3] O. Bahat and A. Makowski, "Optimal replacement policies for nonuniform cache objects with optional eviction," in *INFOCOM*, 2003.

[4] N. Beckmann and D. Sanchez, "A cache model for modern processsors," Massachusetts Institute of Technology, 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 *Proc. HPCA-21*, 2015.

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

[7] P. Billingsley, *Probability and measure*. Wiley, 2008.

[8] B. D. de Dinechin, R. Ayrignac, P.-E. Beaucamps, P. Couvert, B. Ganne, P. G. de Massas, F. Jacquet, S. Jones, N. M. Chaisemartin, F. Riss *et al.*, "A clustered manycore processor architecture for embedded and accelerated applications," in *Proc. HPEC*, 2013.

[9] J. Doweck, "Inside the CORE microarchitecture," in *Proc. HotChips-18*, 2006.

[10] N. Duong, D. Zhao, T. Kim, R. Cammarota, M. Valero, and A. V. Veidenbaum, "Improving cache management policies using dynamic reuse distances," in *Proc. MICRO-45*, 2012.

[11] A. Jaleel, K. B. Theobald, S. C. Steely Jr, and J. Emer, "High performance cache replacement using re-reference interval prediction," in *Proc. ISCA-37*, 2010.

[12] D. Jiménez, "Insertion and promotion for tree-based pseudolru last-level caches," in *Proc. MICRO-46*, 2013.

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

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

[15] S. Khan, Y. Tian, and D. Jiménez, "Sampling dead block prediction for last-level caches," in *Proc. MICRO-43*, 2010.

[16] S. Khan, Z. Wang, and D. Jimenez, "Decoupled dynamic cache segmentation," in *Proc. HPCA-18*, 2012.

[17] A. Kolobov, "Planning with markov decision processes: An ai perspective," *Synthesis Lectures on Artificial Intelligence and Machine Learning*, vol. 6, no. 1, 2012.

[18] N. Kurd, S. Bhamidipati, C. Mozak, J. Miller, T. Wilson, M. Nemani, and M. Chowdhury, "Westmere: A family of 32nm IA processors," in *Proc. ISSCC*, 2010.

[19] R. Mattson, J. Gecsei, D. Slutz, and I. Traiger, "Evaluation techniques for storage hierarchies," *IBM Sys. J.*, vol. 9, no. 2, 1970.

[20] G. Monahan, "A survey of partially observable markov decision processes," *Management Science*, vol. 28, no. 1, 1982.

[21] G. Pekhimenko, T. Huberty, R. Cai, O. Mutlu, P. P. Gibbons, M. A. Kozuch, and T. C. Mowry, "Exploiting compressed block size as an indicator of future reuse," in *Proc. HPCA-21*, 2015.

[22] G. Pekhimenko, V. Seshadri, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Base-delta-immediate compression: practical data compression for on-chip caches," in *Proc. PACT-21*, 2012.

[23] P. Petoumenos, G. Keramidas, and S. Kaxiras, "Instruction-based reuse-distance prediction for effective cache management," in *Proc. SAMOS*, 2009.

[24] M. Puterman, *Markov decision processes: discrete stochastic dynamic programming*. Wiley, 2009.

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

[26] M. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive insertion policies for high performance caching," in *Proc. ISCA-34*, 2007.

[27] E. Rotem, A. Naveh, A. Ananthakrishnan, D. Rajwan, and E. Weissmann, "Power-management architecture of the intel microarchitecture code-named sandy bridge," *IEEE Micro*, vol. 32, no. 2, 2012.

[28] D. Sanchez and C. Kozyrakis, "Zsim: fast and accurate microarchitectural simulation of thousand-core systems," in *Proc. ISCA-40*, 2013.

[29] J. Shun, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, A. Kyrola, H. V. Simhadri, and K. Tangwongsan, "Brief announcement: the problem based benchmark suite," in *Proc. SPAA*, 2012.

[30] M. Takagi and K. Hiraki, "Inter-reference gap distribution replacement: an improved replacement algorithm for set-associative caches," in *Proc. ICS-18*, 2004.

[31] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi, "CACTI 5.1," HP Labs, Tech. Rep. HPL-2008-20, 2008.

[32] R. Wang and L. Chen, "Futility scaling: High-associativity cache partitioning," in *Proc. MICRO-47*, 2014.

[33] C.-J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. C. Steely Jr, and J. Emer, "Ship: signature-based hit predictor for high performance caching," in *Proc. MICRO-44*, 2011.

[34] T. Yoshida, M. Hondou, T. Tabata, R. Kan, N. Kiyota, H. Kojima, K. Hosoe, and H. Okano, "SPARC64 XIfx: Fujitsu's Next Generation Processor for HPC," *IEEE Micro*, no. 2, 2015.

# APPENDIX

## A. COUNTEREXAMPLE TO PREDICTING TIME UNTIL REFERENCE

Fig. 12 lays out a detailed example that shows evicting lines based on their expected time until preference is suboptimal. It shows a particular case where doing so lowers the cache's hit rate. This example demonstrates that cases given in Sec. 3 can arise in practice. It is somewhat contrived to make it as simple as possible, but the central problem arises in real programs as well.

We consider a cache consisting of a single line that has the option of bypassing an incoming accesses. The replacement policy always chooses between two candidates: *(i)* the currently cached line at age $a$, or *(ii)* the incoming reference. Hence all deterministic replacement policies degenerate to choosing an age at which they evict lines, $a_E$.

Moreover, it is trivial to compute the cache's hit rate as a function of $a_E$. All reuse distances less than or equal to $a_E$ hit, and all others are evicted. The hit rate is:[7]

$$\mathrm{P}[\text{hit}] = \frac{\mathrm{P}[D \le a_E]}{a_E \cdot \mathrm{P}[D > a_E] + \sum_{a=1}^{a_E} a_E \cdot \mathrm{P}[D = a]} \tag{19}$$

We consider Eq. 19 on an access stream with a trimodal reuse distance distribution that takes three distinct values at different probabilities. (Three values are needed to construct a counterexample; predicting time until reference is optimal on bimodal distributions.) The example we choose is shown in Fig. 12a. This reuse distance distribution is:

$$\mathrm{P}[D = d] = \begin{cases} 0.25 & \text{If } d = 4 \\ 0.25 & \text{If } d = 8 \\ 0.5 & \text{If } d = 32 \end{cases}$$

The sensible age at which to evict lines are thus either 4, 8, or 32. Fig. 12b computes the hit rate for each. The best choice is to evict lines after 8 accesses, i.e. $a_E = 8$. Fig. 12c shows the expected time until reference for new accesses and those at age 4. Accesses have a lower expected time until reference when new than at age 4, so predicted time until reference would choose to evict at age 4 (i.e., $a_E = 4$). This choice degrades the hit rate.

The reason why predicting time until reference takes this decision is that it incorrectly uses the long reuse distance (i.e., 32) when considering candidates at age 4. This is incorrect because if the cache does the right thing—evict lines at $a_E = 8$—then lines will never make it to age 32. Hence, the replacement policy should ideally not even consider reuse distance 32 when making decisions.

If the maximum age reached in the cache is 8, then only events that occur before age 8 should be taken under consideration. Our contribution is to produce a framework that generalizes to these cases.

[7] Eq. 19 is quite similar to the model PDP [10] uses to determine protecting distances, but note that this formula does not generalize well to caches with more than one line [4].

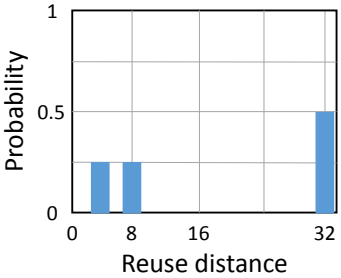
## B. QUANTIFYING MODEL ERROR

The iid reuse distance model assumes each reference's reuse distance is distributed independently of the others, but this is not so. We can get a sense for the magnitude of the error through some simple calculations. We will focus on a single set. The error arises because hits to different lines are disjoint events, but the our model treats them as independent. Hence the number of hits in the iid model follows a binomial distribution [7], and multiple hits can occur on a single access. Suppose the hit rate per line is a constant $g$, the cache size is $N$, and the associativity is $W$. In reality, the miss probability is $1 - Wg$ and the hit probability for a single set is $Wg$. But using a binomial distribution, the miss probability is $(1-g)^W$ and single hit occurs with probability $Wg(1-g)^{W-1}$. Hence, the models do not agree on the probabilities of these events. The error term is the "missing probability" in the binomial model, i.e. the probability of more than one hit. To quantify this error, consider the linear approximation of the binomial distribution around $g = 0$:

$$(1-g)^W = 1 - Wg + O(g^2)$$
$$Wg(1-g)^{W-1} = Wg + O(g^2)$$

When $g$ is small, $O(g^2)$ is negligible, and the binomial distribution approximates the disjoint event probabilities. Since the average per-line hit rate is small, at most $1/N \ll 1/W$, the error from assuming independence is negligible.

**Figure 12: An example where replacing lines by their expected time until reference leads to sub-optimal hit rate. *(a)* Reuse distance probability distribution. *(b)* Hit rate of different policies; retaining lines for 8 accesses is optimal. *(c)* Expected time until reference for different lines; the informal principle says to evict lines after 4 accesses (since 25.33... > 19), and is not optimal.**




(a) Reuse distance distribution

| **Policy** Retain for... | **Hit rate** Hit prob. / Avg. lifetime |
|---|---|
| 4 accesses | $\frac{1/4}{4} = \frac{1}{16}$ |
| 8 accesses | $\frac{1/2}{4\times1/4+8\times3/4} = \frac{1}{14}$ |
| 32 accesses | $\frac{1}{4\times1/4+8\times1/4+32\times1/2} = \frac{1}{19}$ |

(b) Hit-rate of policies

| | **Expected time until reference** |
|---|---|
| **New access** | $4\times1/4+8\times1/4+32\times1/2 = 19$ |
| **Line at age 4** | $(8\times1/3+32\times2/3)-4 = 25.33\ldots$ |

(c) Expected time until reference

## C. EVA ON COMPRESSED CACHES

EVA is easily extended to compressed caches, where candidates have different sizes. We call this policy C-EVA. The basic idea is to compute the opportunity cost *per byte*, rather than per line as in uncompressed caches. Hence, if $B$ is the size of the cache in bytes, the basic formula for EVA (Eq. 3) is modified to:

$$h(a) = \mathrm{P}\big[\text{hit}|\text{age } a\big] - \frac{\text{Hit rate}}{B} \times \mathrm{E}\big[L - a|\text{age } a\big] \times \text{Line size}$$

Additionally, CAMP [21], the state-of-the-art replacement policy for compressed caches, adapts its policy to different compressed block sizes. The intution behind this is that compressed size is correlated with how the data is used, so classifying by compressed size can help make better replacement decisions.

We also extend EVA with classification by compressed size. We classify compressed sizes logarithmically, so with a line size of 64 B, C-EVA has six different size classes. For simplicity, we assume that candidates' size does not change, although this is not always true (a line's size can change when it is written). With this modification, it is easy to compute the per-class EVA, similar to Sec. 4.2.