Maximizing Cache Performance Under Uncertainty

Nathan Beckmann*
Carnegie Mellon University
beckmann@cs.cmu.edu

Daniel Sanchez
Massachusetts Institute of Technology
sanchez@csail.mit.edu

Abstract*—Much prior work has studied cache replacement, but a large gap remains between theory and practice. The design of many practical policies is guided by the optimal policy, Belady's MIN. However, MIN assumes perfect knowledge of the future that is unavailable in practice, and the obvious generalizations of MIN are suboptimal with imperfect information. *What, then, is the right metric for practical cache replacement?

We propose that practical policies should replace lines based on their economic value added (EVA), the difference of their expected hits from the average. Drawing on the theory of Markov decision processes, we discuss why this metric maximizes the cache's hit rate. We present an inexpensive implementation of EVA and evaluate it exhaustively. EVA outperforms several prior policies and saves area at iso-performance. These results show that formalizing cache replacement yields practical benefits.

I. INTRODUCTION

Last-level caches consume significant resources, often over 50% of chip area [24], 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 policy is Belady's MIN [10, 25], which evicts the candidate referenced furthest in the future. But MIN needs perfect knowledge of the future, which makes it impractical. In practice, policies must cope with *uncertainty*, never knowing exactly when candidates will be referenced. Theory-based policies account for uncertainty by using a simplified, statistical model of the reference stream in which the optimal policy can be solved for.

Unfortunately, these policies generally perform poorly compared to empirical designs. The key challenge faced by theoretical policies is choosing their underlying statistical model. This model should capture enough information about the access stream to make good replacement decisions, yet must be simple enough to analyze. For example, some prior work uses the independent reference model (IRM), which assumes that candidates are referenced independently with static, known probabilities. In this model, the optimal policy is to evict the candidate with the lowest reference probability, i.e., LFU [2]. Though useful in other areas (e.g., web caches [4]), 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 [11, 14, 16, 17, 19, 20, 21, 30, 35, 39]. We observe that, unlike the IRM, these policies do not assume static reference probabilities. Instead, they *exploit dynamic behavior* through various 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.

This paper seeks to bridge theory and practice. We take a principled approach that builds on insights from recent empirical designs. First, we show that policies should replace candidates by their *economic value added* (EVA); i.e., how many more hits one expects from each candidate vs. the average candidate. 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, and describe how EVA reconciles them in a single, intuitive metric.
- We show that EVA maximizes the cache hit rate by drawing on Markov decision process (MDP) theory.
- We present a practical implementation of EVA, which we have synthesized in a 65 nm commercial process. EVA adds 1% area on a 1 MB cache vs. SHiP. Our implementation is the first adaptive policy that does not require set sampling or auxiliary tag monitors.
- We evaluate EVA against prior high-performance policies on SPEC CPU2006 and OMP2012 over many cache sizes. EVA reduces LLC misses over these policies at equal area, closing 57% of the gap from random replacement to MIN vs. 47% for SHiP [39], 41% for DRRIP [17], and 42% for PDP [14]. *Fewer misses translate into large area savings—EVA matches SHiP's performance with gmean 8% less total cache area.*

These contributions show that formalizing cache replacement yields practical benefits. EVA blends theory and practice to maximize upon available information and outperform many recent empirical policies. Beyond our particular design, we expect our analysis will prove useful in the design of future high-performance policies.

Road map: Sec. II reviews prior approaches to cache replacement, and Sec. III discusses the key constraints that practical policies face. Sec. IV presents EVA and Sec. V sketches its theoretical justification. Sec. VI presents a simple implementation, which Sec. VII evaluates.

II. BACKGROUND

Practical 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 [10, 25]).

Broadly speaking, prior work has taken two approaches to replacement under uncertainty. On the one hand, designers

*This work was done while the author was at MIT.

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, designers 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 longstanding approach takes inspiration from theory (i.e., MIN) and has been implemented in recent empirical policies [17, 20]. (However, Sec. III shows that this strategy is suboptimal.)

Yet despite the evident synergy between theory and practice, the vast majority of research has been on the empirical side. We believe that, given the diminishing performance improvements of recent empirical policies, theoretical approaches deserve a second look.

Replacement in theory: The challenge for theoretical policies is to define a reference model that is simple enough to solve for the optimal policy, yet accurate enough that this policy is effective. In 1971, Aho et al. [2] 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 is to evict the page with the lowest reference probability, i.e., LFU. Though the IRM ignores temporal locality, and recent work recognizes this shortcoming, it nevertheless remains the de facto model [37].

However, the IRM is an especially poor reference model for processor caches. Unlike web caches, which are accessed by thousands of independent users, processor caches are accessed by relatively few threads. As a result, their references tend to be tightly correlated and exhibit complex, dynamic behaviors. It is this time-varying complexity, which must be captured to achieve good performance, that the IRM discards.

For example, consider a program that scans repeatedly 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: reference probabilities are not static, but change in correlated and predictable patterns. The independent reference model does not capture this common behavior.

More recently, a large body of work [1, 9, 15, 18, 33] has studied cache behavior under more complex memory reference models. (Garetto et al. [15, §VI] summarize recent work that models caches in distributed systems.) However, since these reference models are difficult to analyze, prior work focuses on comparing policies (e.g., LRU vs. k-LRU) or tuning their parameters, not on finding policies that truly maximize upon available information.

Since the behavior of real programs is not captured in analytically tractable models, empirical work has tended to ignore theory and focus on best-effort heuristics instead.

Replacement in practice: Many high-performance policies try to emulate MIN through various heuristics. DIP [30] avoids thrashing by inserting most lines at low priority, and detects when it is advantageous to do so. SDBP [21] and PRP [11] predict which lines are unlikely to be reused. RRIP [17, 39] and IbRDP [27] try to predict candidates' times until reference. PDP [14] protects lines from eviction for a fixed number of accesses. IRGD [35] computes a statistical cost function from sampled reuse distance histograms. And Hawkeye [16] emulates MIN's past decisions. Without a theoretical foundation, it is unclear if any of these policies takes the right approach. Indeed, no policy dominates across benchmarks (Sec. VII), 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* to select a victim, most commonly by using the *recency* and *frequency* heuristics [28]. Most policies employ some form of *recency*, favoring candidates that were referenced recently: e.g., LRU uses recency alone, and RRIP [17, 39] predicts a longer time until reference for older candidates. Similarly, a few policies that do not assume recency still base their policy on when a candidate was last referenced: PDP [14] protects candidates until a certain age; and IRGD [35] uses a heuristic function of ages. Another common way policies expoit dynamic behavior is through *frequency*, favoring candidates that were previously reused: e.g., LFU uses frequency alone, and "scan-resistant" policies like ARC [26] and SRRIP [17] favor candidates that have been reused at least once.

Second, recent high-performance policies *adapt themselves* to the access stream to varying degrees. DIP [30] detects thrashing with set dueling, and thereafter inserts most lines at LRU to prevent thrashing. DRRIP [17] inserts lines at medium priority, promoting them only upon reuse, and avoids thrashing using the same mechanism as DIP. SHiP [39] extends DRRIP by adapting the insertion priority based on the memory address, PC, or instruction sequence. Likewise, SDBP [21], PRP [11], and Hawkeye [16] learn the behavior of different PCs. And PDP [14] and IRGD [35] use auxiliary monitors to profile the access pattern and periodically recompute their policy.

These two trends show that *(i)* candidates reveal important information over time, and *(ii)* policies should learn from this information by adapting themselves to the access pattern. 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., recency or frequency). Rather, it is a general approach that aims to make the best use of limited information, so that prior heuristics arise naturally when appropriate (Sec. IV).

III. Replacement under uncertainty

Given the many approaches taken in prior work, we start from first principles. All replacement policies have the same goal and face the same constraints: they try to maximize the

cache's hit rate with limited cache space. We can develop the intuition behind EVA by precisely characterizing these tradeoffs.

We are interested in where hits come from and how cache space is used over time. To analyze this, we break up cache space over time into *lifetimes*, the idle periods between hits and evictions. In Fig. 1, a line comes into the cache at access t, hits at access $t+4$, and is evicted at $t+10$. This gives two lifetimes, as shown. We further define a line's *age* as the number of accesses since it was last referenced. So in Fig. 1, the line's first lifetime ends in a hit at age 4 (there are 4 accesses from t to $t+4$), and the second lifetime ends in an eviction at age 6 (there are 6 accesses from $t+4$ to $t+10$). Note how age resets to zero upon each reference.

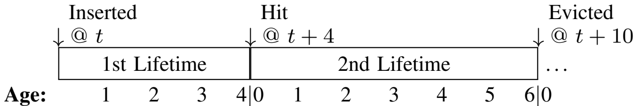


Fig. 1: *Lifetimes* and *ages* for a single cache line over time (increasing left-to-right). Time is measured in accesses.

Lifetimes and ages let us precisely describe the tradeoffs in cache replacement. We introduce two random variables, H and L, which give the probability of hits and lifetimes of different ages. That is, $\mathbb{P}[H=5]$ is the probability that a line will hit at age 5, and $\mathbb{P}[L=5]$ is the probability that its lifetime will end at age 5 (i.e., in either a hit or eviction at age 5). Probability is a natural way to reason about the inherent uncertainty facing all practical replacement policies.

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

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

but are constrained by limited cache space. Specifically, the average lifetime equals the cache size, N:

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

Eq. 2 is essentially Little's Law with an arrival rate of one, since time is measured in accesses (see [5, §B.3] for its derivation).

Comparing these two equations, we see that hits are equally 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.

But how should policies balance these competing, incommensurable objectives? With perfect information, MIN is a simple policy that achieves both ends. Unfortunately, MIN does not easily generalize under uncertainty: obvious generalizations like evicting the candidate with the highest expected time until reference [17,27,35] or the lowest expected hit probability [11,21] are inadequate, since *they only account for one side of the tradeoff.*

Example: Inspired by MIN, several recent policies predict time until reference and evict the candidate with the longest one [17, 27, 35]. A simple counterexample shows why predictions of time until reference are inadequate.

Suppose the replacement policy has to choose between two candidates: A is referenced immediately with probability $9/10$, and in 100 accesses with probability $1/10$; and B is always referenced every two accesses. (See [5, §B.2] for an example of how such situations could arise in practice.) 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 $9/10$, since every access to A has a $9/10$ probability of hitting. In contrast, evicting A yields an expected hit rate of $1/2$, since accesses to B produce one hit every two accesses. 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, say, by keeping A for one access to see if it hits. But simple generalizations of MIN ignore this insight, e.g., expected time until reference is unduly 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, so it is wrong for it to influence replacement decisions.

IV. EVA REPLACEMENT POLICY

Since it is inadequate to consider either hit probability or time until reference alone, we must find some way to reconcile them in a single metric. In general, the optimal metric should satisfy three properties: *(i)* it considers only future behavior, since the past is a sunk cost; *(ii)* it prefers candidates that are more likely to hit; and *(iii)* it penalizes candidates that take longer to hit. These properties both maximize the hit probability and minimize time spent in the cache, as desired.

We achieve these properties 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". EVA is essentially a cost-benefit analysis about whether a candidate's odds of hitting are worth the cache space it will consume.

This section describes EVA and gives a motivating example. Sec. V discusses why EVA maximizes the hit rate.

Intuition: EVA views each replacement candidate as an investment, trying to retain the candidates that yield the highest profit (measured in hits). First, EVA rewards each candidate for its expected future hits. Then, since cache space is a scarce resource, EVA needs to account for how much space each candidate will consume. EVA does so by "charging" each candidate for the time it will spend in the cache. Specifically, EVA charges candidates at a rate of a single 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. Altogether, the EVA for a candidate is:

$$\boxed{\text{EVA} = \text{Expected hits} - \frac{\text{Cache hit rate}}{\text{Cache size}} \times \text{Expected time}}$$

A. Computing EVA

We compute EVA using conditional probability and the random variables H and L introduced above. (Sec. VI discusses how to efficiently sample these distributions.) To begin, we compute EVA from candidates' ages, using the time since last reference to inform EVA's decisions. Inferring EVA from candidates' ages roughly corresponds to the recency heuristic (Sec. II), except that EVA does *not* assume recently used candidates are more valuable (see example below).

We compute EVA from the candidate's expected remaining lifetime and its hit probability in that lifetime. Let $r(a)$ be the expected hit probability (reward, or benefit) and $c(a)$ the forgone hits (cost). Then from the equation above, EVA at age a is just their difference, $\text{EVA}(a) = r(a) - c(a)$.

The reward $r(a)$ is the expected number of hits for a line of age a, i.e., its hit probability. This is basic conditional probability, where age a restricts the sample space to lifetimes at least a accesses long:

$$r(a) = \mathbb{P}\big[\text{hit}|\text{age } a\big] = \frac{\mathbb{P}[H > a]}{\mathbb{P}[L > a]} \tag{3}$$

Likewise, the forgone hits $c(a)$ is the expected number of hits that could be obtained from replacing the candidate, which increases in proportion to the candidate's remaining lifetime. Each access would yield, on average, hits equal the cache's hit rate h divided by its size N:

$$\begin{aligned} c(a) &= \frac{h}{N} \times \mathbb{E}\big[L - a|\text{age } a\big] \\ &= \frac{h}{N} \times \frac{\sum_{x=1}^{\infty} x \cdot \mathbb{P}[L = a + x]}{\mathbb{P}[L > a]} \end{aligned} \tag{4}$$

To summarize, we select a victim by comparing each candidate's EVA and evict the candidate with the lowest EVA. Our implementation does *not* compute EVA during replacement. Instead, it infrequently ranks ages by computing their EVA. Moreover, Eqs. 3 and 4 at age a each require just a few arithmetic operations to compute from age $a+1$, so computing EVA is inexpensive. Sec. VI gives the full details.

B. Example

To see how EVA works, consider an example application that scans alternating over two arrays of different sizes, called the 'small' and 'big' arrays. The cache size is such that the small array fits in the cache, but the big array does not. Specifically, the arrays take 16 K and 128 K lines, and the cache has 64 K lines. Fig. 2 shows the resulting distribution of *reuse distance*, i.e., the number of references between accesses to the same address. (Reuse distances are twice the size of arrays because each receives half of accesses.) Without further information about candidates, this distribution is all the replacement policy has to choose among candidates. What is the right policy?

Replacement policy: In this example, we want to cache the small array and some part of the big array. Fig. 3 shows how EVA ranks candidates to achieve this.

Fig. 3 shows a candidate's EVA (y-axis) vs. its age (x-axis). Initially, at age zero, candidates have decent EVA because half the time they are part of the small array and will hit quickly. To be precise, at age zero a candidate's EVA is zero, since we

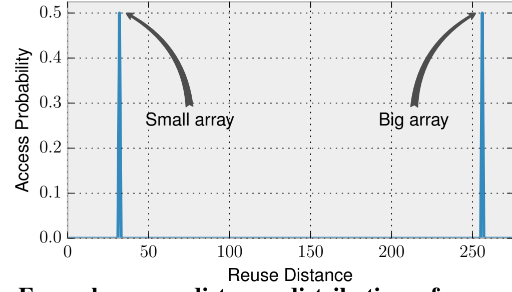


Fig. 2: Example reuse distance distribution of an application scanning over two arrays. The big array does not fit in cache.

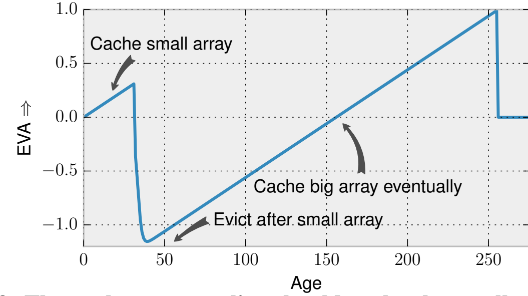


Fig. 3: The replacement policy should cache the small array and part of the big array. Due to uncertainty, it must wait until after the small array to evict candidates.

have not yet learned anything about the candidate. Zero is not a low EVA value; it indicates average behavior.

A candidate's EVA increases with age as it gets closer to (possibly) hitting in the small array. This is because the hit probability does not change, but its expected remaining lifetime shrinks in proportion to age. The result is a line increasing from age zero up to the reuse distance of the small array ("Cache small array" in Fig. 3). The replacement policy will thus evict MRU among ages in this range, since younger lines have lower EVA. This is the optimal policy for a scanning pattern.

However, if a candidate's age increases beyond the small array, then its EVA plummets because the policy has learned it must be an access to the big array ("Evict after small array" in Fig. 3). These candidates have many more accesses to go before hitting (i.e., a long expected remaining lifetime), so their EVA is negative—they take so long to hit that they are not worth the investment. The replacement policy will thus preferentially evict candidates with ages in this range, i.e., accesses to the big array that still have a long time to hit.

Finally, EVA gradually increases with age as a candidate gets closer to hitting in the big array, until eventually old candidates have the highest EVA among all candidates ("Cache big array eventually" in Fig. 3). Their EVA is larger than young candidates because their hit probability is larger, since some young candidates are accesses to the big array that are later evicted. The replacement policy will thus protect older candidates from eviction, fitting as much of the big array as possible.

Performance: Fig. 4 shows the performance of a cache using this policy, namely its distribution of hits and evictions at different ages. All accesses to the small array hit, but only a fraction of those to the big array hit. Evictions are concentrated after the small array, since these ages have the lowest EVA

in Fig. 3. This is as desired: the cache cannot fit both arrays, and the policy must wait until after the small array to learn whether an access is to the big or small array.

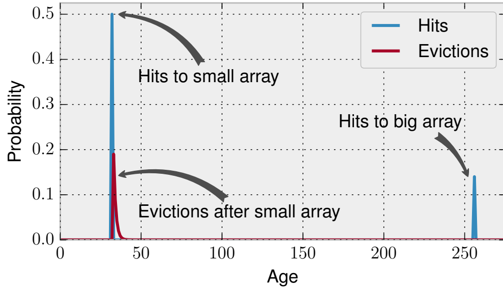


Fig. 4: The cache holds the entire small array and part of the big array (hit rate: 64%). Uncertainty wastes some cache space, since evictions must occur after the small array.

Uncertainty comes at a cost, since these evictions waste space. Ideally, a cache with 64 K lines could hold the 16 K lines of the small array plus 48 K lines of the big array, giving a hit rate of $1/2 \times 16\,\mathrm{K}/16\,\mathrm{K} + 1/2 \times 48\,\mathrm{K}/128\,\mathrm{K} = 69\%$. But with uncertainty, some space must be spent learning whether accesses are to the small or big array. This wastes space proportional to the size of the small array, so given the information in Fig. 2, Eq. 2 implies that the maximum achievable hit rate on this example is 64%. EVA achieves this performance, maximizing upon the available information.

In contrast, no "protecting distance" in PDP [14] can do so. Protecting the small array gives a hit rate of 50%, and protecting the big array wastes so much cache space that it actually lowers the hit rate to 44%.

Summary: This example shows how EVA changes over time, adapting its policy to the access pattern. Our contribution is identifying hit probability and expected lifetime as the key tradeoffs in cache replacement, and reconciling them in a single metric.

For simplicity, we have so far assumed that information is limited to that revealed by aging. We now discuss how EVA uses more information to make better decisions.

C. EVA with classification

The main challenge that replacement policies face is *uncertainty* about candidates' future behavior. One common technique to reduce uncertainty is to break candidates into multiple *classes* and tune the policy to each class. Classification is widely used in prior work, e.g., policies classify by reuse, thread, PC, etc. (Sec. II).

Since EVA is based on conditional probability, EVA naturally supports classification by conditioning on a candidate's class. (Indeed, EVA thus far simply classifies candidates by age.) This formal probabilistic approach lets EVA specialize its policy to each class without any a priori preferences among classes.

Example: We can see how EVA incorporates classification by revisiting the above example. In this application, we should differentiate accesses to the big and small arrays. This lets us rank the candidates of each class separately, as illustrated in Fig. 5. Classification changes the ranks at small ages, as the replacement policy no longer needs to wait to learn about candidates. This has two implications: *(i)* accesses to the small array become more valuable because they are certain to hit quickly, and *(ii)* accesses to the big array have low EVA at age zero, since we now know *immediately* that they will take many accesses to hit. These differences let the replacement policy evict candidates more quickly, reducing waste from evictions and thereby caching more of the big array. In this example classification gives perfect information, but generally this is not the case.

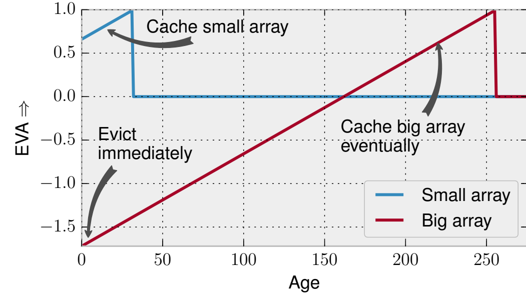


Fig. 5: Classification reduces uncertainty. Now, the replacement policy need not wait until after the small array to evict.

Classification improves cache performance, as shown in Fig. 6. There are now more hits to the big array, and the hit rate improves to 69%, which is *optimal* for this cache size and access pattern. This performance improvement is possible because evictions now occur very early, so minimum cache space is wasted on them.

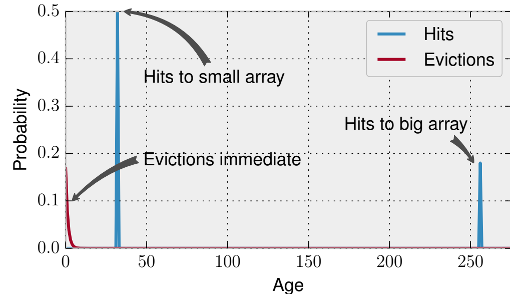


Fig. 6: With classification, the cache holds the entire small array and more of the big array (hit rate: 69%). Evictions occur immediately, freeing space for more hits.

To illustrate classification in EVA, we now show how EVA incorporates the frequency heuristic by distinguishing candidates that have been reused at least once, similar to prior scan-resistant policies (Sec. II).

Classification by reuse: We divide the cache into two classes: lines that have hit at least once, and newly inserted lines that have not. We use this simple scheme because it has proven effective in prior work [17, 22, 26]. (In the above example, the "reused class" is the small array and the fraction of the big array that fits.) However, unlike prior work, EVA does not assume an a priori preference for reused candidates, and EVA supports arbitrary classification schemes (e.g., our technical report [6] classifies candidates by size in compressed caches).

Until now, we have considered only the current lifetime when computing expected future hits, as we lacked further information. When distinguishing among classes, this is insufficient, and we must consider all future lifetimes since classes often behave differently over the long run.

The details depend on the classification scheme. When classifying by reuse, if a lifetime ends in a hit, then the next will be a reused lifetime (by definition). Otherwise, if it ends in an eviction, then the next will be a non-reused lifetime (again by definition). Fig. 7 illustrates classification by reuse; we want to compute EVA over all future hits and misses, starting from a line of age a and class C on the left.

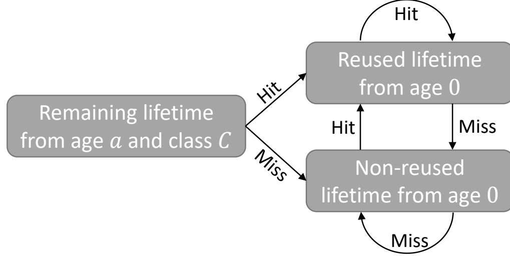


Fig. 7: With classification, we must consider behavior in future lifetimes to compute EVA.

Details: We denote the EVA of reused lines as $\text{EVA}^R(a)$, and the EVA of non-reused lines as $\text{EVA}^{NR}(a)$. We further condition EVA's terms with a line's class. For example, the reward for a reused line is:

$$r^R(a) = \frac{\mathbb{P}\big[H > a | \text{reused}\big]}{\mathbb{P}\big[L > a | \text{reused}\big]} \tag{5}$$

Compared to Eq. 3, the only difference is the added condition of "reused". Forgone hits ($c^R(a)$) and non-reused lines ($r^{NR}(a)$, $c^{NR}(a)$) are similarly conditioned.

We refer to class C whenever either R or NR apply. EVA for a single lifetime is unchanged: $\text{EVA}^C_{\text{lifetime}}(a) = r^C(a) - c^C(a)$, but now future lifetimes cannot be ignored.

We can express long-run EVA for any age as a function of the long-run EVA of reused and non-reused lines at age zero, following Fig. 7. If the hit rate of class C at age a is $h^C(a)$, then the corresponding, long-run EVA is:

$$\begin{aligned}\text{EVA}^C(a) = &\,\text{EVA}^C_{\text{lifetime}}(a) && \text{(Current lifetime)}\\ &+ h^C(a)\cdot\text{EVA}^R(0) && \text{(Hits} \rightarrow \text{Reused)}\\ &+ (1 - h^C(a))\cdot\text{EVA}^{NR}(0) && \text{(Misses} \rightarrow \text{Non-reused)}\end{aligned}$$

Finally, the average access's EVA—i.e, the "average difference from the average"—is zero by definition, so:

$$0 = h \cdot \text{EVA}^R(0) + (1 - h)\cdot\text{EVA}^{NR}(0)$$

Solving these equations reveals a simple relationship between each class's EVA and the EVA of reused lines:

$$\text{EVA}^C(a) = \text{EVA}^C_{\text{lifetime}}(a) + \tfrac{h^C(a) - h}{1 - h^R(0)}\cdot\text{EVA}^R_{\text{lifetime}}(0) \tag{6}$$

The second term essentially says that the more hits class C produces vs. the average, the more it will behave like a reused line in the future.

Summary: Classification is a common way to reduce uncertainty. EVA naturally supports classification through conditional probability, automatically specializing its policy to each class. Though conceptually simple, we must account for long-term differences between classes to accurately compute EVA.

We focus on classification by reuse for concreteness, but these ideas apply to other classification schemes as well. One must simply express how lines transition between classes and then solve for EVA. For example, our technical report [6] classifies candidates in a compressed cache by their size.

EVA takes a different approach to classification than most recent work. Several recent policies break accesses into many classes, often using the requesting PC, and then adapt their policy to each class [11, 16, 21, 39]. However, with thousands of classes, these policies are restricted to simple decisions for each class (e.g., a single bit—is the class is valuable or not?), and they rely on a baseline policy (e.g., random, LRU, or RRIP) once candidates are inserted. In contrast, EVA ranks ages at fine granularity, but this restricts EVA to use fewer classes (e.g., just two with classification by reuse). An important area of future work is to explore the best balance of these approaches.

V. Why EVA is the right metric

The previous section described and motivated EVA. This section discusses why EVA is the right metric to maximize cache performance under uncertainty. We first present a naïve metric that intuitively maximizes the hit rate, but show that it unfortunately cannot capture long-run behavior. Fortunately, prior work in Markov decision processes (MDPs) has encountered and solved this problem, and we show how EVA adapts the MDP solution to cache replacement.

Naïve metric: We want a replacement metric that maximizes the cache's hit rate. With perfect knowledge, MIN achieves this by greedily "buying hits as cheaply as possible," i.e. by keeping the candidates that are referenced in the fewest accesses. Stated another way, MIN retains the candidates that get the most hits-per-access. Therefore, a simple metric that might generalize MIN is to predict each candidate's hits-per-access, or hit rate:

$$\mathbb{E}[\text{hit rate}] = \lim_{T\to\infty} \frac{\text{Expected hits after } T \text{ accesses}}{T} \tag{7}$$

and retain the candidates with the highest hit rate. Intuitively, keeping these candidates over many replacements tends to maximize the cache's hit rate.

The problem: Eq. 7 suffices when considering a single lifetime, but it cannot account for candidates' long-run behavior over many lifetimes: To estimate the future hit rate, we must compute expected hits over arbitrarily many future accesses. However, as cache lines are replaced many times, they tend to converge to average behavior because their replacements are generally unrelated to their original contents. Hence, all candidates' hit rates converge in the limit. In fact, all candidates' hit rates are *identical*: solving Eq. 7 as $T \to \infty$ yields h/N, the cache's per-line hit rate, for all ages and classes.

In other words, Eq. 7 loses its discriminating power over long time horizons, degenerating to random replacement. So while estimating candidates' hit rates is intuitive, this approach is fundamentally flawed as a replacement metric.

The solution in a nutshell: EVA sidesteps this problem by changing the question. Instead of asking *which candidate gets a higher hit rate?*, EVA asks *which candidate gets more hits?*

The idea is that Eq. 7 compares the long-run hit rates of retaining different candidates:

$$\lim_{T\to\infty} \frac{\text{hits}_1(T)}{T} \overset{?}{>} \lim_{T\to\infty} \frac{\text{hits}_2(T)}{T}, \tag{8}$$

but unfortunately both sides of the comparison converge to the same quantity. We can avoid this problem by simply multiplying by T and subtracting both sides:

$$\lim_{T\to\infty}\left[\text{hits}_1(T)-\text{hits}_2(T)\right] \overset{?}{>} 0 \tag{9}$$

This equation is equivalent to the original, but measures small differences in long-run behavior that would otherwise disappear in the limit.

These metrics are are useful in different contexts. When rates are unequal in the limit, Eq. 8 is well-behaved, but Eq. 9 diverges to ∞. Eq. 8 is thus the appropriate metric. However, when rates are the same in the limit, Eq. 8 cannot discriminate, but Eq. 9 can. Hence, Eq. 9 is the appropriate metric. Since hit rates are equal in the limit, it makes sense to follow Eq. 9 and rank candidates by which gets more hits in the limit.

The solution in detail: We are not the first to encounter this optimization problem or devise the above solution, which has been previously proposed in the context of Markov decision processes (MDPs). MDPs model decision-making under uncertainty by extending Markov chains to have *actions* in each state. Each action yields an associated *reward*.[1]

A large body of prior work has studied policies to achieve different objectives and proved conditions of optimality. (For a comprehensive treatment, see Puterman [29].) In our case, the reward is cache hits, the actions are accesses (some of which invoke the replacement policy), and we care about maximizing the *average reward* (i.e., hit rate). When studying the optimal policy for such MDPs, researchers ran into exactly the same problem as above: all actions yield the same average reward in the limit. To solve this problem, they introduce a new quantity called the *bias*, which is similar to Eq. 9 above.

The bias is defined as the expected difference in total reward from the current state vs. the average. In other words, after T actions from the current state, one would expect some total reward. Meanwhile, averaging across states, the MDP would have yielded a reward equal to its average reward times T. The bias is just the difference between these two quantities in the limit:

$$\text{Bias} = \lim_{T\to\infty} \text{Reward after } T \text{ actions} - \text{Avg. reward} \times T$$

This bias is thus similar to Eq. 9, but it compares actions against the average rather than against each other.

Fig. 8 illustrates the bias. The y-axis shows total reward, and the x-axis shows time (measured in actions). The solid line shows the expected total reward starting from the current state; the dashed line shows the expected total reward averaged over all states. The expected reward fluctuates initially, as returns from the current state differ from the average, but eventually settles to yield the average reward in the limit. The distance between these two lines in the limit is the bias, generally a finite but non-zero number.

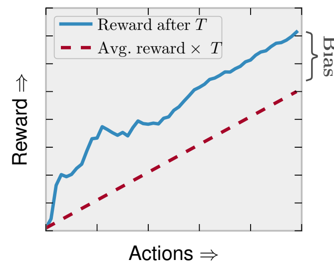


Fig. 8: Bias is the long-run difference between the total reward and the average reward.

[1] For interested readers, our technical report [6] presents a detailed cache replacement MDP using a reference model from our recent work [8, 9].

Maximizing the bias provably maximizes the average reward [29, §8.4]. Thus, to maximize the cache's hit rate, we should try to maximize the bias among replacement candidates.

From theory to practice: EVA does exactly this. Bias and EVA are directly analogous:

$$\text{Bias} = \lim_{T\to\infty} \text{Reward after } T - \text{Avg. reward} \times T$$
$$\downarrow \qquad\qquad \downarrow \qquad\qquad \downarrow \qquad\qquad \downarrow$$
$$\text{EVA} = \text{Expected hits} - \frac{\text{Hit rate}}{\text{Cache size}} \times \text{Time}$$

Sec. IV-A gives the bias over a single lifetime: the reward is the hit probability, the average reward is a single line's hit rate, and the time horizon T is the remaining lifetime. Sec. IV-C gives the bias over all future lifetimes (i.e., as $T \to \infty$), as illustrated in Fig. 7. MDP theory tells us how to combine these terms into a metric that maximizes the cache's hit rate.

Two critical points deserve emphasis:

Optimality: First, *maximizing the bias provably maximizes the average reward*. It is not a heuristic. MDP theory provides metrics that allow greedy, local decisions to provably optimize global outcomes [29]. Maximizing the bias is equivalent to maximizing the hit rate, except that it is well-behaved in the limit and so lets us model long-run behavior.

Generality: Second, because maximizing the bias reconciles hit probability and cache space, which we have argued are the fundamental constraints (Sec. III), it gives broad optimality conditions. For example, both perfect information (MIN) and the IRM [2] are special cases.

EVA is easy to extend to many different contexts. For instance, with small modifications EVA can model a compressed cache, where candidates have different sizes and policies based on time until reference—even MIN!—are clearly suboptimal [6].

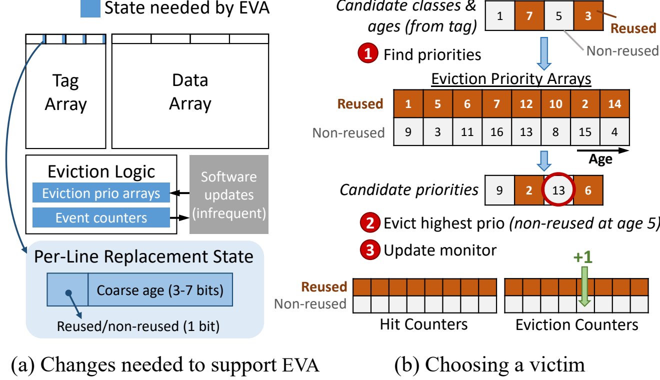
Convergence: MDPs are solved using iterative algorithms that are guaranteed to converge to an optimal policy within arbitrary precision. In particular, *policy iteration* [23] alternates between computing the expected total reward for each state and updating the policy. Our implementation takes an analogous approach, alternatively monitoring the cache ("computing rewards") and updating ranks.

EVA can converge to changes in behavior because it is neutral towards candidates it knows nothing about—their EVA is zero. Since zero represents average behavior, other, below-average candidates will be evicted preferentially. EVA thus naturally explores the behavior of these "unknown candidates", and a few tend to age long enough to discover any changes in their behavior. If even a few of these candidates hit, conditional probability gives them large EVA because so few candidates survive to these ages. This creates a virtuous circle, as favoring these candidates leads to further hits.

Nevertheless, with classification, it is possible to construct cases where EVA gets stuck evicting one class. This pathology could be fixed by rarely tagging some candidates as "explorers" that are not evicted until they reach the maximum age. However, we find that EVA performs well in practice and do not consider this mechanism further.

VI. Implementation

Our implementation is shown in Fig. 9: *(i)* A small table, called the *eviction priority array*, ranks candidates to 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 updates the eviction priorities.



(a) Changes needed to support EVA (b) Choosing a victim

Fig. 9: An efficient implementation of EVA.

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 [7]; however, we also describe a hardware-only implementation in Sec. VI-D.

Unlike prior policies, EVA does not devote a fraction of sets to monitoring alternative policies (cf. [17, 30]), nor does it require auxiliary tags to monitor properties independent of the replacement policy (cf. [7, 14, 35]). As a result, our implementation makes full use of the entire cache and eliminates overheads from monitors. EVA also does not need to forward the requesting PC to the LLC, which, though widely used in prior work, complicates the core-cache interface and adds network bandwidth.

A. Hardware operations

Aging: We use per-set, coarsened ages [14]. 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 $2^k - 1$.

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_1 is ranked higher than age a_2, then $\text{EVA}(a_1) < \text{EVA}(a_2)$ (as described below). 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.

Algorithm 1. Algorithm to compute EVA and update ranks.

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

1: **for** $a \leftarrow 2^k$ **to** 1 **:** ▷ Compute hit rates from counters.
2: **for** $c \in \{NR, R\}$ **:**
3: hits$_c$ += hitCtrs$[c, a]$
4: events$_c$ += hitCtrs$[c, a]$ + evictionCtrs$[c, a]$
5: $h^R[a] \leftarrow$ hits$_R$/events$_R$
6: $h^{NR}[a] \leftarrow$ hits$_{NR}$/events$_{NR}$
7: $h \leftarrow$ (hits$_R$ + hits$_{NR}$)/(events$_R$ + events$_{NR}$)
8: perAccessCost $\leftarrow h \times A/N$
9: **for** $c \in \{NR, R\}$ **:** ▷ Compute EVA (Eqs. 3 & 4).
10: expLifetime, hits, events $\leftarrow 0$
11: **for** $a \leftarrow 2^k$ **to** 1 **:**
12: expectedLifetime += events
13: eva$[c, a] \leftarrow$ (hits − perAccessCost × expectedLifetime)/events
14: hits += hitCtrs$[c, a]$
15: events += hitCtrs$[c, a]$ + evictionCtrs$[c, a]$
16: evaReused $\leftarrow$ eva$[R, 1]/(1 - h^R[0])$ ▷ Differentiate classes.
17: **for** $c \in \{NR, R\}$ **:**
18: **for** $a \leftarrow 2^k$ **to** 1 **:**
19: eva$[c, a]$ += $(h^C[a] - h) \times$ evaReused
20: order $\leftarrow$ ArgSort(eva) ▷ Finally, rank ages by EVA.
21: **for** $i \leftarrow 1$ **to** 2^{k+1} **:**
22: rank[order[i]] $\leftarrow 2^{k+1} - i$
23: **return** rank

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 $k = 7$.

B. Software updates

The eviction priority array is a versatile mechanism that can implement many policies, specifically any *ranking function* of ages [9], such as random, LRU, PDP [14], IRGD [35], etc. We set priorities to implement EVA.

Periodically (every 256 K accesses), an OS runtime computes EVA for active applications. First, we read the hit and lifetime distributions (H and L) 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.)

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. Algorithm 1 gives an efficient procedure to compute Eq. 6. Updates occur in four passes over ages. In the first pass, we compute hit rates h^R, h^{NR}, and h by summing counters. Second, we compute per-lifetime EVA incrementally in five additions, one multiplication, and one division per age. Third, classification 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 (excluding debugging and comments), incurs negligible runtime overheads (see below), and can be found online at `http://people.csail.mit.edu/sanchez`.

C. 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 [36]

	Area		Energy	
	(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. VI-D)	0.052	0.30%	380 / 128 K	0.1%

TABLE I: Implementation overheads at 65 nm.

for all SRAMs (using register files instead of SRAM makes the circuit 4× larger). Table I 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 [13] and energy from CACTI. Overheads are small, totaling 0.2% area and 1.0% energy. 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 [13].

Software updates: Updates complete in a few tens of K cycles and run every 256 K LLC accesses (i.e., several M cycles). Specifically, with $k = 7$ age bits, updates take 43 K cycles on an Intel Xeon E5-2670. 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.

We consider these overheads negligible and choose $k = 7$, but EVA lets designers trade off overheads and performance. For example, choosing $k = 5$ reduces software overheads by three-quarters while sacrificing little performance (Fig. 14).

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.

Our evaluation shows that EVA is competitive with prior policies when using fewer age bits. *But 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. 14). EVA trades off larger tags for improved performance, making better use of the 99% of cache area not devoted to replacement, and thus 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 and implementation complexity. In hardware, EVA adds only narrow increments and comparisons; in software, EVA adds a short, low-overhead reconfiguration procedure (Algorithm 1).

D. 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 [38]), or on systems with dedicated OS cores (e.g., the Kalray MPPA-256 [12] or Fujitsu Sparc64 XIfx [40]).

However, if software updates are undesirable, we have also implemented and synthesized a custom microcontroller that performs updates in hardware. 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 $1052\,\text{B}$ with $k = 7$. We have also implemented a small FSM to compute the eviction priorities, adding $2 \times 2^{k+1} \times (k+1)$ bits, or $512\,\text{B}$.

This microcontroller adds small overheads (Table I)—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 [31]). So although we believe software updates are a simpler design, EVA can be implemented entirely in hardware if desired.

Cores	Westmere-like OOO [32] 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 II: Configuration of the simulated systems for single- (ST) and multi-threaded (MT) experiments.

VII. Evaluation

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

A. Methodology

We use zsim [32] to simulate systems with 1 and 8 OOO cores with parameters shown in Table II. We simulate LLCs 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 [34]), and with a stream prefetcher validated against real Westmere systems [32].

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 per tag and $\epsilon = 1/32$ [17]. SHiP uses $M = 2$ bits and PC signatures with idealized, large history counter tables [39]. DRRIP is only presented in text because it performs similarly to SHiP, but occasionally slightly worse. These policies' performance degrades with larger M (Fig. 14). 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. We evaluate performance and area because the power overheads from replacement are negligible—e.g., EVA's are 0.01% of TDP (Sec. VI-C). Area and performance are thus the main design constraints.

For each LLC size, we use CACTI to model data and tag area. We use 45 tag bits for address and coherence state. We add

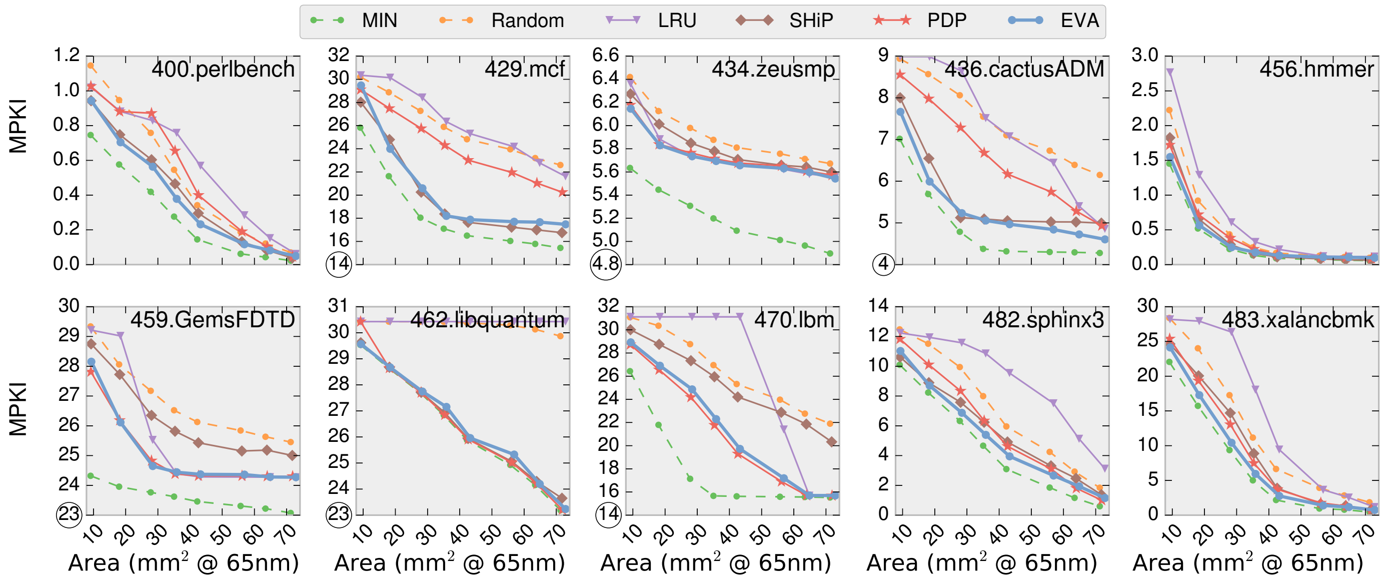


Fig. 10: Misses per thousand instructions (MPKI) vs. total cache area across sizes (1 MB–8 MB) for MIN, random, LRU, SHiP, PDP, and EVA on selected memory-intensive SPEC CPU2006 benchmarks. (Lower is better.)

replacement tag bits and other overheads taken from prior work. Whenever unclear, we use favorable numbers for other policies: DRRIP and SHiP add 2 bits/tag, and SHiP adds 1.875 KB for tables (0.1 mm^2). PDP adds 3 bits/tag and 10 K NAND gates (0.02 mm^2). LRU uses 8 bits/tag. Random adds no overhead. Since MIN is our upper bound, we also grant it zero overhead. Finally, EVA adds 8 bits/tag and 0.04 mm^2 (Sec. VI).

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 [3], we instrument SPEC OMP2012 apps with heartbeats that denote application-level work. 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. *EVA's performance results include the time spent in software updates, which is negligible (Sec. VI-C).*

B. Single-threaded results

Fig. 10 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 generally maximizes upon available information, so the right replacement strategies naturally emerge where appropriate. As a result, EVA successfully captures the benefits of SHiP and PDP within a common framework, and sometimes outperforms both. Since EVA performs consistently well, and 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`).

EVA edges closer to optimal replacement: Fig. 11 compares the practical policies against MIN, showing the average MPKI gap over MIN across the most memory-intensive SPEC CPU2006 apps[2]—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. 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%.

[2] All with $\geq$3 L2 MPKI; Fig. 10 plus `bzip2`, `gcc`, `milc`, `gromacs`, `leslie3d`, `gobmk`, `soplex`, `calculix`, `omnetpp`, and `astar`.

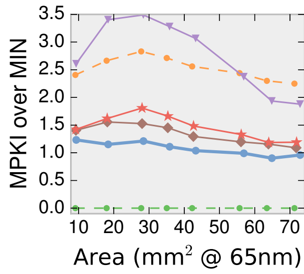


Fig. 11: MPKI above MIN for each policy on SPEC CPU2006.

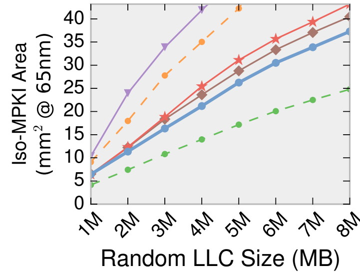


Fig. 12: Iso-MPKI cache area: avg MPKI equal to random at different LLC sizes.

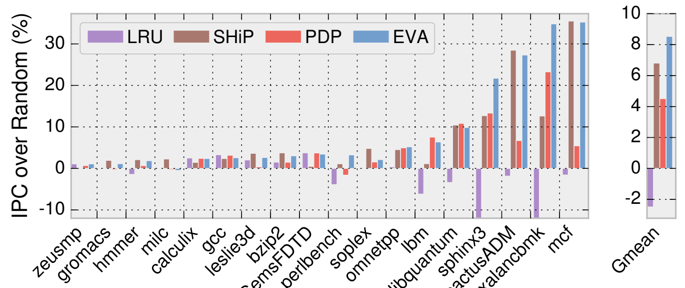


Fig. 13: Performance across SPEC CPU2006 apps on a 35 mm² LLC. (Only apps with >1% difference shown.)

EVA saves cache space: Because EVA improves performance, it needs less cache space than other policies to achieve a given level of performance. Fig. 12 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 mm^2 EVA cache achieves the same MPKI as a 4 MB cache using random replacement, whereas SHiP needs 23.6 mm^2 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, though some performance gap is unavoidable due to the costs of uncertainty (Sec. IV-B).

EVA achieves the best end-to-end performance: Fig. 13 shows the IPC speedups over random replacement at 35 mm^2, 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.

EVA achieves consistently good speedups across apps, whereas 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. 14 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.

Unlike prior policies, EVA's performance improves steadily

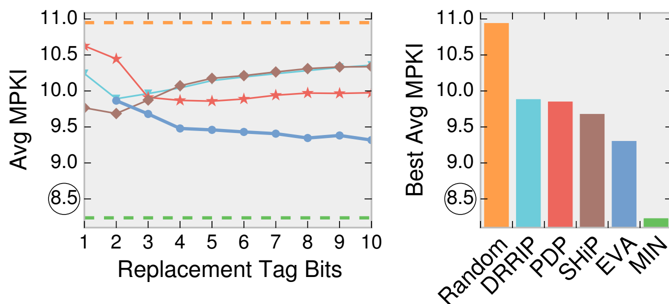


Fig. 14: Avg MPKI for different policies at 4 MB vs. tag overheads (lower is better).

with more state, and its peak performance exceeds prior policies by a good margin. With 2 bits, EVA performs better than PDP, similar to DRRIP, and slightly worse than SHiP. 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. 12).

With very few age bits, no single choice of age granularity A works well for all applications. To make EVA perform well with few bits, software adapts the age granularity using a simple heuristic: 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, then decrease A by one. We find this heuristic rapidly converges to the right age granularity across all evaluated applications. Only Fig. 14 uses this heuristic, and *it is disabled for all other results.* Since EVA is most area-efficient with larger tags, the design we advocate (8 b tags) does not employ this heuristic.

C. Multi-threaded results

Fig. 15 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

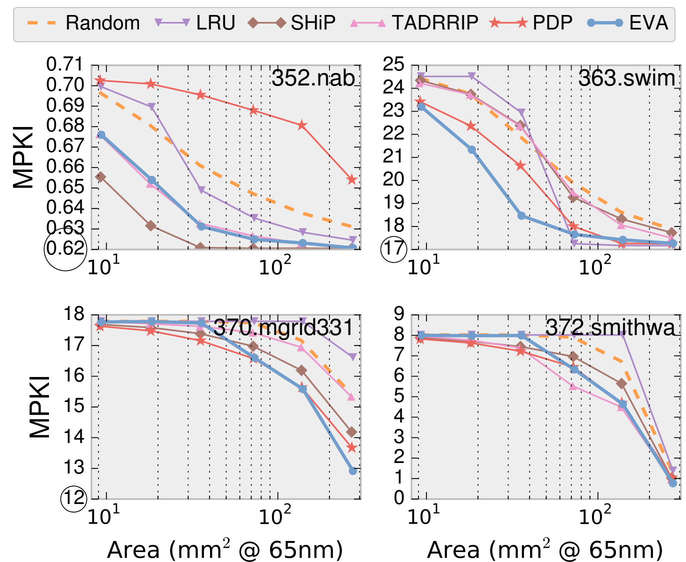


Fig. 15: MPKI vs. total cache area across sizes (1 MB, 2 MB, 4 MB, 8 MB, 16 MB, and 32 MB) for different policies on selected SPEC OMP2012 apps. (Lower is better.)

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 apps from the remainder.

As in 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 (Fig. 15 plus `md`, `botsspar`, and `kd`), 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%.

VIII. Conclusion

The key challenge faced by practical replacement policies is how to cope with uncertainty. Simple approaches like predicting time until reference are flawed. We have argued for replacement by *economic value added* (EVA), starting from first principles and drawing from prior planning theory. We further showed that EVA can be implemented with trivial hardware, and that it outperforms existing high-performance policies nearly uniformly on single- and multi-threaded benchmarks.

Acknowledgments

We thank David Karger for pointing us towards Markov decision processes. We thank Harshad Kasture, Po-An Tsai, Joel Emer, Guowei Zhang, Suvinay Subramanian, Mark Jeffrey, Anurag Mukkara, Mor Harchol-Balter, Haoxian Chen, and the anonymous reviewers for their helpful feedback. This work was supported in part by NSF grant CCF-1318384 and a grant from the Qatar Computing Research Institute.

References

[1] M. Ahmed, S. Traverso, P. Giaccone, E. Leonardi, and S. Niccolini, “Analyzing the performance of LRU caches under non-stationary traffic patterns,” *arXiv:1301.4909v1 [cs.NI]*, 2013.

[2] A. Aho, P. Denning, and J. Ullman, “Principles of optimal page replacement,” *J. ACM*, 1971.

[3] A. Alameldeen and D. Wood, “IPC considered harmful for multiprocessor workloads,” *IEEE Micro*, vol. 26, no. 4, 2006.

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

[5] N. Beckmann, “Design and analysis of spatially-partitioned shared caches,” Ph.D. dissertation, Massachusetts Institute of Technology, 2015.

[6] N. Beckmann and D. Sanchez, “Bridging theory and practice in cache replacement,” Massachusetts Institute of Technology, Tech. Rep. MIT-CSAIL-TR-2015-034, 2015.

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

[8] N. Beckmann and D. Sanchez, “Cache calculus: Modeling cache performance through differential equations,” *IEEE CAL*, 2016.

[9] N. Beckmann and D. Sanchez, “Modeling cache performance beyond lru,” in *Proc. HPCA-22*, 2016.

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

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

[12] B. D. de Dinechin, R. Ayrignac, P.-E. Beaucamps, P. Couvert, B. Ganne, P. G. de Massas *et al.*, “A clustered manycore processor architecture for embedded and accelerated applications,” in *Proc. HPEC*, 2013.

[13] J. Doweck, “Inside the CORE microarchitecture,” in *HotChips-18*, 2006.

[14] 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.

[15] M. Garetto, E. Leonardi, and V. Martina, “A unified approach to the performance analysis of caching systems,” *ACM TOMPECS*, vol. 1, no. 3, 2016.

[16] A. Jain and C. Lin, “Back to the future: Leveraging Belady’s algorithm for improved cache replacement,” in *Proc. ISCA-43*, 2016.

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

[18] P. Jelenković and A. Radovanović, “Least-recently-used caching with dependent requests,” *Theor. Comput. Sci.*, vol. 326, no. 1, 2004.

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

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

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

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

[23] A. Kolobov, “Planning with Markov decision processes: An AI perspective,” *Synthesis Lectures on Artificial Intelligence and Machine Learning*, vol. 6, no. 1, 2012.

[24] N. Kurd, S. Bhamidipati, C. Mozak, J. Miller, T. Wilson, M. Nemani *et al.*, “Westmere: A family of 32nm IA processors,” in *Proc. ISSCC*, 2010.

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

[26] N. Megiddo and D. S. Modha, “ARC: A self-tuning, low overhead replacement cache.” in *Proc. FAST*, 2003.

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

[28] S. Podlipnig and L. Böszörmenyi, “A survey of web cache replacement strategies,” *ACM Computing Surveys (CSUR)*, 2003.

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

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

[31] 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.

[32] D. Sanchez and C. Kozyrakis, “ZSim: fast and accurate microarchitectural simulation of thousand-core systems,” in *Proc. ISCA-40*, 2013.

[33] R. Sen and D. A. Wood, “Reuse-based online models for caches,” in *ACM SIGMETRICS Performance Evaluation Review*, 2013.

[34] J. Shun, G. Blelloch, J. Fineman, P. Gibbons, A. Kyrola, H. V. Simhadri *et al.*, “Brief announcement: the problem based benchmark suite,” in *Proc. SPAA*, 2012.

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

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

[37] S. Traverso, M. Ahmed, M. Garetto, P. Giaccone, E. Leonardi, and S. Niccolini, “Temporal locality in today’s content caching: why it matters and how to model it,” in *Proc. SIGCOMM*, 2013.

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

[39] 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.

[40] T. Yoshida, M. Hondou, T. Tabata, R. Kan, N. Kiyota, H. Kojima *et al.*, “SPARC64 XIfx: Fujitsu’s Next Generation Processor for HPC,” *IEEE Micro*, vol. 30, no. 2, 2015.