10.48047/jocaaa.2024.33.06.201

Cache Replacement Policies for Modern Multicore Systems: From LRU to Mockingjay

[1]Purnendu Das, [2,*]Bishwa Ranjan Roy
[1,2]Department of Computer Science, Assam University, Silchar.

***Corresponding Author:** [2]*brroy88@gmail.com

***Abstract*—**The replacement policy of a last-level cache decides which resident block is sacrificed when a miss must be filled. That apparently local choice has system-wide consequences in a modern multicore processor: it changes memory-level parallelism, DRAM traffic, prefetch usefulness, interference among applications, throughput, and fairness. This survey organizes the development of processor cache replacement from least-recently used (LRU) and its practical approximations through adaptive insertion, re-reference interval prediction, dead-block and reuse prediction, and Belady-inspired learning, ending with Mockingjay. Rather than listing policies chronologically, the paper compares what each policy predicts, the granularity at which it predicts, how it converts prediction into insertion or eviction decisions, its metadata and timing implications, and its behavior under multicore interference. Reported performance figures are taken from the original publications and are not treated as directly comparable because cache sizes, workloads, prefetchers, and metrics differ. The survey shows a clear progression: policies first protected recency, then learned when to distrust recency, next predicted whether a block would be reused, and finally estimated the relative time of future reuse. It concludes with practical selection guidance and open problems involving quality of service, prefetch/coherence interaction, heterogeneous cores, chiplets, security, and reproducible evaluation.

***Index Terms*—**cache replacement, multicore processors, last-level cache, LRU, RRIP, dead-block prediction, Hawkeye, Mockingjay.

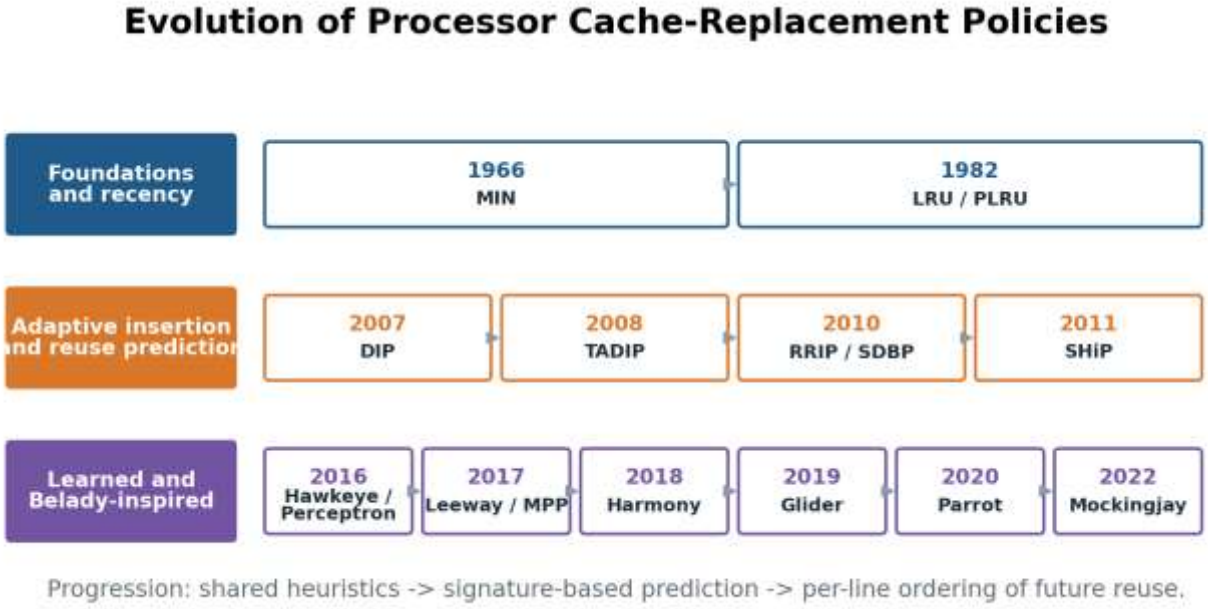

Fig. 1. Evolution of the policy families covered in this survey. The timeline is a conceptual organization of the literature, not a claim that every intermediate policy is shown.

10.48047/jocaaa.2024.33.06.201

I. INTRODUCTION

Processor performance increasingly depends on avoiding long-latency accesses beyond the on-chip cache hierarchy. In an associative cache, a miss to a full set creates a deceptively simple question: which resident line should be evicted? The answer is difficult because the replacement controller sees only the past, while the correct decision depends on future accesses. Belady's MIN policy provides the offline optimum by evicting the block whose next use is farthest in the future [1]. MIN is an indispensable analytical reference, but it requires knowledge that hardware cannot possess.

LRU converts the unknown future into a locality heuristic: the line unused for the longest time is assumed least valuable [2], [3]. This approximation is effective when recent reuse predicts near-future reuse, and it is sufficiently simple to explain and validate. However, scans, cyclic working sets slightly larger than the cache, phase changes, and irregular graph-style accesses can make recency actively misleading. Moreover, exact LRU metadata and update logic become costly at high associativity, motivating pseudo-LRU (PLRU), not-recently-used (NRU), and random replacement in practical designs.

Multicore processors amplify every weakness. A shared LLC multiplexes capacity among threads with different footprints and reuse characteristics. One thread can evict another thread's high-value lines, a prefetcher can insert blocks that displace useful demand data, and the globally best victim may not be the victim that preserves fairness or a service-level objective. Replacement is therefore entangled with partitioning, insertion, bypass, promotion, and prefetch management. Utility-based cache partitioning (UCP) [4] addresses allocation explicitly; PIPP [7] integrates partitioning with promotion and insertion. This survey keeps those mechanisms in view while concentrating on the policy that ranks candidate victims.

The central argument of the survey is that modern policies differ less by their final victim-selection rule than by the *future quantity they try to infer*. DIP asks which insertion behavior is currently better [5]. RRIP estimates a coarse re-reference interval [8]. SDBP and SHiP predict deadness or reuse from sampled histories and signatures [9], [10]. Hawkeye learns binary cache friendliness from an offline-optimal model reconstructed over the past [11]. Perceptron and multiperspective predictors combine multiple contextual signals [12], [14]. Mockingjay estimates reuse distance and maintains an estimated time remaining for each line [18]. Figure 3 later presents this common prediction-to-action pipeline.

This paper makes four contributions. First, it provides a mechanism-centered taxonomy covering the requested lineage from LRU through Mockingjay. Second, it treats multicore interference, fairness, and prefetch interaction as first-class design constraints rather than separate extensions. Third, it compiles reported results and hardware properties while explicitly warning against cross-paper numerical ranking. Fourth, it offers workload-to-policy guidance and identifies research questions that remain open even after Belady-inspired learning.

II. BACKGROUND AND EVALUATION CRITERIA

A. Cache Actions and the Replacement Critical Path

A conventional set-associative LLC performs tag lookup, detects a hit or miss, and on a miss identifies an invalid entry or a victim way. A replacement policy may intervene at four points. It can bypass the incoming block, choose its insertion position or prediction state, promote or demote a line after a hit, and rank lines at eviction. These controls are coupled. A line inserted with high priority is effectively protected; a line bypassed is treated as immediately dead; a promotion policy determines how quickly one hit erases prior evidence.

The design must satisfy a tight timing and storage budget. Per-line state scales with LLC capacity; per-set state scales with set count; signature tables scale with the number and width of contexts; sampled structures trade training coverage for cost. Update bandwidth also

10.48047/jocaaa.2024.33.06.201

matters because a policy may need to touch metadata on every LLC access even though victim selection occurs only on misses. Consequently, a more accurate predictor can lose practical value if it adds latency, consumes excessive SRAM, creates high write activity, or is difficult to verify.

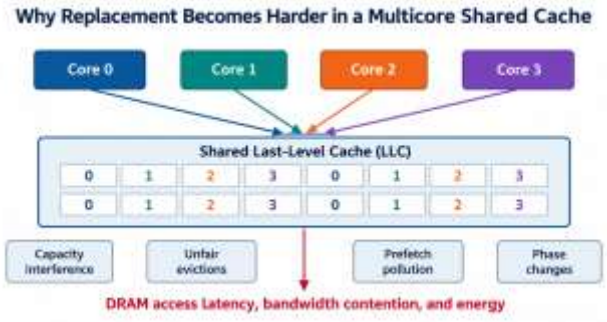

Fig. 2. A shared LLC couples the access streams of multiple cores. Replacement must absorb capacity interference, unfair evictions, prefetch pollution, and phase changes before a miss reaches DRAM.

B. What "Better" Means in a Multicore LLC

Miss rate and misses per kilo-instruction (MPKI) are necessary but incomplete. A saved miss has greater performance value when it lies on the critical path and cannot overlap with other misses. Multicore papers therefore commonly report weighted speedup, throughput, harmonic speedup, maximum slowdown, or unfairness in addition to MPKI. A replacement policy can improve aggregate throughput while harming a latency-sensitive co-runner, so performance and fairness should be reported together. UCP's marginal-utility reasoning [4] illustrates why the same number of ways has different value for different applications.

The offline reference also changes with the system objective. Classical MIN minimizes misses for a single sequence [1]. In a multicore interleaving, the global sequence depends on relative progress: changing the cache policy changes each core's speed and therefore changes the future interleaving. Mockingjay explicitly notes that MIN is not cleanly defined for a multicore system [18]. Likewise, prefetches blur the meaning of a hit because a demand miss can be converted into prefetch traffic rather than eliminated. Harmony rethinks Belady-style optimality for demand and prefetched accesses [15].

C. Comparison Method Used in This Survey

The paper compares policies along six axes: prediction target, context or signature, granularity, replacement action, adaptation mechanism, and implementation cost. Reported numerical results are quoted only as evidence that a mechanism was effective in its own experimental environment. They are not normalized into a synthetic ranking. Different papers use different processor widths, cache capacities, associativities, memory systems, benchmark versions, core counts, warm-up methods, prefetchers, and performance metrics; a one-point difference across papers is therefore not meaningful.

TABLE I: MECHANISM-CENTERED COMPARISON OF MAJOR POLICY FAMILIES

Family	Predicted quantity	Granularity	Primary action	Central trade-off
LRU / PLRU	Recency order	Line / set	Evict least recent; approximate ordering	Strong baseline; scans and cyclic thrashing
DIP / TADIP	Best insertion mode	Set; per-core selector	Choose LRU-like or bimodal insertion	Adapts to scans; limited signal
SRRIP / DRRIP	Re-reference interval class	Per line	Insert/promote/evict by RRPV	Compact; coarse ordinal prediction

10.48047/jocaaa.2024.33.06.201

Family	Predicted quantity	Granularity	Primary action	Central trade-off
SDBP	Dead versus live	PC-indexed predictor	Bypass or deprioritize dead blocks	Sampling accuracy and aliasing
SHiP	Signature reuse tendency	PC/region/sequence signature	Initialize RRIP state by confidence	Robust, low-cost reuse correlation
Hawkeye	Cache-friendly versus averse	PC + sampled OPTgen	Protect friendly; evict averse	Binary labels lose ordering detail
Perceptron / MPP	Weighted reuse evidence	Multiple hashed features	Bypass and/or rank by score	Rich context; higher metadata
Leeway	Live distance	Line + PC predictor	Dynamic reuse- or bypass-oriented mode	Handles variability across generations
Harmony	Demand/prefetch utility	Belady-derived classes	Prefetch-aware ordering	Depends on prefetch behavior
Mockingjay	Reuse distance / ETA	PC hit/miss signature + line ETR	Bypass; evict farthest from ETA	Fine ordering; sampled training cost

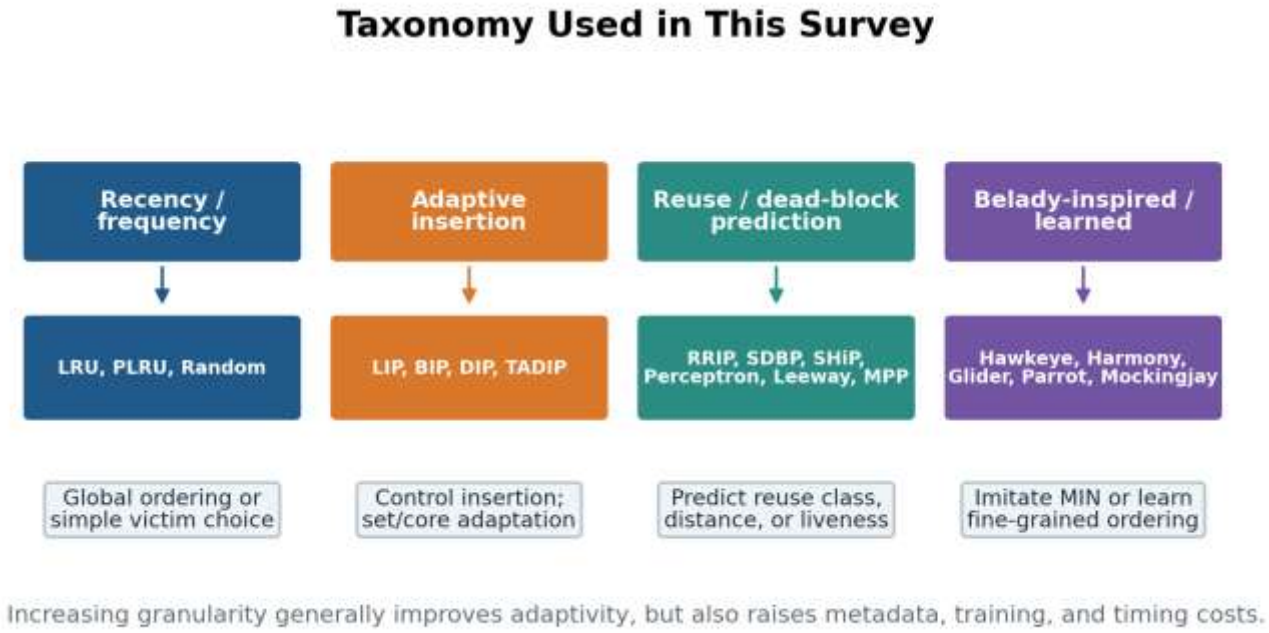

Fig. 3. Taxonomy used in this survey. Greater prediction granularity generally increases adaptability, metadata, and training complexity.

III. CLASSICAL RECENCY AND EARLY MULTICORE MANAGEMENT

A. LRU, PLRU, Random, and Frequency

LRU maintains a total order of the ways in each set. A hit moves the referenced way to the most-recently-used position; a fill inserts a new line at or near that position; a miss evicts the least-recently-used way. Its conceptual appeal follows the stack property analyzed by Mattson et al. [2]: for a fixed reference string, LRU caches of increasing capacity form nested contents, enabling stack-distance analysis. LRU is also deterministic and easy to reason about, which is why it remains the baseline against which sophisticated policies are

10.48047/jocaaa.2024.33.06.201

presented.

Exact LRU is expensive for a highly associative LLC because a hit can require updating several ordering relationships. Tree-PLRU records direction bits in a binary tree and identifies a victim with approximately logarithmic state and logic. NRU records whether a line was used in the current epoch. Random replacement removes recency-update traffic and sometimes avoids adversarial cyclic behavior, but discards valuable locality information. LFU-like policies protect frequently used blocks but adapt slowly to phases unless counts are aged. These alternatives reveal a recurring design principle: replacement metadata is itself a constrained predictor, and simplifying it changes both accuracy and responsiveness.

LRU fails in two canonical patterns. A streaming scan inserts blocks that are touched once, pushing reusable blocks toward eviction. A cyclic sequence whose working set is just larger than associativity can repeatedly evict the next-needed line. Both failures arise because insertion at the MRU position grants an unproven line the same protection as a repeatedly reused line. Adaptive insertion policies attack precisely this assumption.

B. Partitioning Is Related but Not Equivalent

In a shared LLC, replacement chooses among lines that currently occupy a set, while partitioning constrains how much occupancy each thread may obtain. UCP estimates each application's marginal miss reduction from additional ways and allocates ways to maximize aggregate utility [4]. PIPP shows that partitioning alone can be undermined if insertion and promotion fight the target allocation; it therefore uses probabilistic insertion and promotion to move the cache toward desired partitions [7]. These techniques are important context for replacement because no victim ranking can guarantee fairness if one thread is allowed to monopolize occupancy, and no partition can deliver utility if replacement discards the wrong blocks inside the assigned region.

A useful mental model is hierarchical control. The allocator determines *whose* line may be displaced; the replacement policy determines *which* eligible line is least valuable. Modern systems may also impose way masks, classes of service, slice hashing, or QoS quotas. A policy evaluated in an unconstrained shared cache may behave differently inside a partition because its training stream, set pressure, and effective associativity all change.

IV. ADAPTIVE INSERTION AND RE-REFERENCE INTERVALS

A. LIP, BIP, and Dynamic Insertion Policy

The LRU insertion policy (LIP) inserts a new line at the LRU position, giving it only a probationary stay; a hit promotes it to MRU. LIP protects established working sets from scans, but it can adapt too slowly when new lines truly have high reuse. The bimodal insertion policy (BIP) mostly inserts at LRU but occasionally inserts at MRU, allowing a new working set to enter gradually. DIP uses set dueling to choose dynamically between conventional LRU insertion and BIP [5]. A small number of leader sets are dedicated to each candidate policy; a saturating policy-selection counter records which leaders incur fewer misses; follower sets use the current winner.

DIP's lasting contribution is not only BIP, but the hardware methodology of sampling alternatives online. The original study reported a 21% average MPKI reduction for a 1-MB, 16-way cache and closed roughly two thirds of the gap between LRU and an offline optimum, with less than two bytes of global selection state in its simplified selector [5]. Those figures are configuration-specific, but the mechanism is general: use a small controlled experiment inside the cache to decide which global heuristic matches the current phase.

B. Thread-Aware DIP

TADIP extends the selection logic to multicore workloads [6]. A single global selector can be dominated by one application and apply the wrong insertion mode to another. TADIP

10.48047/jocaaa.2024.33.06.201

associates selection state with threads so that each core can prefer LRU-like or bimodal insertion while sharing the same physical sets. The paper reported throughput gains over LRU across 2-, 4-, 8-, and 16-core configurations, with fewer than two bytes of selector state per core [6]. The deeper lesson is that adaptation scope must match heterogeneity: a per-cache decision is insufficient when co-runners have different reuse distributions.

C. SRRIP, BRRIP, and DRRIP

RRIP replaces an exact recency stack with a small ordinal prediction of when a line will be referenced again [8]. Each line stores a re-reference prediction value (RRPV). Low values indicate likely near reuse; the maximum value identifies an eviction candidate. SRRIP normally inserts at a distant-but-not-immediate state, giving new lines a chance without allowing them to displace the working set aggressively. Hits move lines toward near reuse. BRRIP uses bimodal insertion, placing most lines at the distant state and a small fraction closer. DRRIP uses set dueling to select between SRRIP and BRRIP.

With two bits per line, RRIP can represent four reuse classes and avoid the update complexity of exact LRU. It is resistant to scans because most incoming lines are not granted maximum protection, and it is resistant to cyclic thrashing because a hit can sharply improve a line's state. The original paper reported average throughput improvements over LRU for both single-core and four-core studies and emphasized lower implementation complexity than exact LRU or LFU [8]. RRIP became a substrate for later predictors: rather than redesigning victim selection, a predictor initializes or modifies the RRPV.

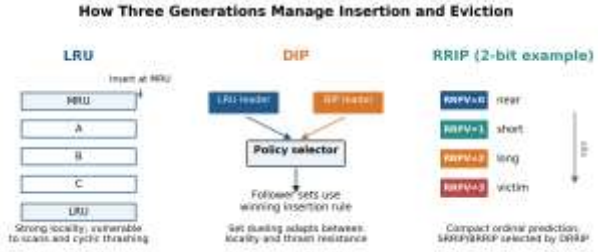

Fig. 4. LRU ranks by recency, DIP adapts insertion through set dueling, and RRIP stores a compact ordinal estimate of re-reference distance.

V. REUSE AND DEAD-BLOCK PREDICTION

A. Sampling Dead-Block Prediction

A dead block will not be referenced again before eviction. If the cache can recognize it early, the block can be bypassed or selected immediately, preserving capacity for live blocks. SDBP trains a PC-indexed predictor using sampled sets [9]. Sampled metadata observes whether lines are reused; the predictor generalizes that outcome to nonsampled sets. The policy's key advantage is leverage: a relatively small sampled structure can guide decisions for a much larger LLC.

The SDBP study reported miss reductions of 11.7% for single-thread workloads and 23% for multicore workloads, with speedup improvements of 5.9% and 12.5%, respectively, in its evaluated configurations [9]. It also quantified predictor energy, an important practice because prediction tables can consume a nontrivial fraction of LLC metadata energy. SDBP's main failure modes are false positives, which evict a line that would have been reused, and aliasing, where unrelated PCs share predictor state. A false negative wastes capacity; a false positive can destroy a critical hit, so predictors often use conservative thresholds.

B. SHiP: Signature-Based Hit Prediction

SHiP observes that reuse correlates with context beyond recency [10]. It associates a signature with the instruction PC, memory region, or instruction sequence that inserted a line, and maintains a saturating counter reflecting whether blocks with that signature were reused.

10.48047/jocaaa.2024.33.06.201

The counter chooses the initial RRIP state. Reuse-prone signatures receive protection; low-confidence signatures are inserted near eviction. On eviction, a line that was never reused trains its signature downward; a reused line trains it upward.

This structure separates *prediction* from *replacement representation*: the signature table learns behavior, while RRIP supplies a compact per-line ordering. The paper reported approximately 10% sequential and 11%-12% multiprogrammed performance improvement over LRU in the evaluated systems [10]. SHiP is attractive because it captures stable producer behavior with modest hardware. It struggles when one signature generates multiple object classes or when reuse changes quickly within the same PC, motivating richer features and variability-aware predictors.

C. Perceptron and Multiperspective Prediction

The perceptron reuse predictor treats dead/live classification as a weighted combination of features rather than a lookup keyed by one signature [12]. Candidate features include PC-derived hashes, address information, recency, and access history. Each active feature contributes a signed weight; their sum yields a reuse score. Training adjusts weights toward the observed outcome. This permits the predictor to represent correlations that a single table cannot, while using integer arithmetic suitable for hardware.

The MICRO 2016 study reported a substantially lower false-positive rate than SDBP and SHiP in its multiprogrammed experiments and reported multicore weighted-speedup gains over LRU [12]. Multiperspective reuse prediction (MPP) extends the idea with several perspectives and feature-selection machinery, enabling different contextual views to vote on reuse [14]. MPP reported 9.0% geomean single-thread speedup and 8.3% normalized weighted speedup for multiprogrammed workloads over LRU in its environment [14]. The cost is more tables, more hashing, training logic, and greater susceptibility to implementation details such as feature aliasing and update saturation.

D. Leeway and Generational Variability

Binary dead-block predictors assume a signature has a reasonably stable outcome. In practice, different dynamic instances of the same PC can live for very different durations. Leeway predicts a block's *live distance*: the maximum recency position at which it was previously hit [13]. It then allows a line to remain resident within that predicted leeway. A dynamic mechanism chooses between a reuse-oriented policy, which avoids premature eviction, and a bypass-oriented policy, which attacks dead capacity.

Leeway is significant because it treats variability as a first-class property instead of merely prediction noise. Its paper reports storage of 52.5 KB for a 16-way, 2-MB LLC for the main design and presents a lower-cost NRU variant [13]. Reported gains vary across SPEC and CloudSuite workloads, with especially strong benefits in some data-serving and map-reduce cases. The design illustrates an important general rule: the right predictor is not always the one with the highest average accuracy; it may be the one whose error asymmetry matches the cost of premature eviction versus wasted residency.

10.48047/jocaaa.2024.33.06.201

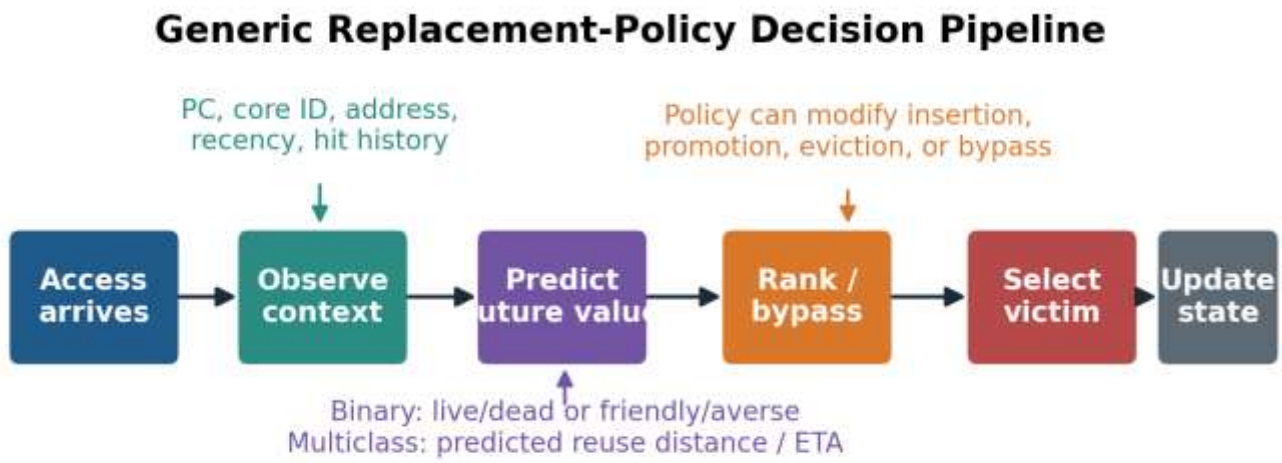

Fig. 5. Generic pipeline shared by modern policies. Policies differ chiefly in the context they observe, the future quantity they predict, and the action through which prediction affects occupancy.

VI. BELADY-INSPIRED AND LEARNED REPLACEMENT

A. Hawkeye: Learning Cache Friendliness from OPTgen

Hawkeye's key insight is to use Belady's algorithm as a teacher over *past* accesses [11]. OPTgen reconstructs, for sampled sets, whether a past access would have been a hit under an optimal policy for a cache of the target capacity. The PC of a cache-friendly access trains a predictor upward; a cache-averse access trains it downward. The main cache then combines this binary classification with RRIP-like victim ranking, preferentially protecting lines produced by friendly PCs and evicting averse lines.

This approach differs from dead-block prediction. A line may be reused but still be cache-averse if retaining it displaces a line with greater value. Conversely, learning from an optimal occupancy model aligns training with the cache's capacity constraint. Hawkeye reported 8.4% improvement over LRU in its single-core evaluation and 15.0% for four-core mixes, compared with 12.0% for the prior state of the art in the paper's multicore setting [11]. Its approximate hardware budget included a sampled OPTgen structure and predictor state; the paper discusses a 16-KB predictor budget in addition to tag-associated state for a 2-MB LLC.

Hawkeye's limitation is its binary output. Two cache-friendly lines can have very different next-use times, yet the classification alone does not establish their relative order. It also relies on sampled histories and on the stability of PC behavior. Later policies retain the Belady-inspired teacher but seek richer ordering information.

B. Harmony: Prefetch-Aware Optimality

A demand-only optimal policy can behave poorly in a prefetched cache because evicting a block changes future prefetch traffic. Harmony formalizes this interaction by distinguishing demand and prefetch intervals and by developing Demand-MIN and Flex-MIN objectives [15]. The practical policy uses Belady-derived insight to decide which prefetched and demand-fetched lines deserve protection. The work demonstrates that replacement and prefetching cannot be optimized independently: a policy must consider whether an evicted line would be demanded, prefetched again, or consume bandwidth before reuse.

This result has broad implications. First, hit rate alone can reward a policy that retains prefetched blocks while increasing bandwidth. Second, a prefetcher changes the apparent reuse signature of a PC; predictor training must distinguish demand hits, prefetch hits, late

10.48047/jocaaa.2024.33.06.201

prefetches, and useless prefetches. Third, policies should be reevaluated when the prefetcher changes because a ranking learned with no prefetching may be wrong under aggressive lookahead.

C. Glider and Parrot: Learning as a Design Instrument

Glider applies deep learning to discover which program histories contain information about Belady-style decisions [16]. Its offline recurrent model is intentionally too expensive for direct hardware deployment; the model is interpreted to derive a practical online predictor using compact integer machinery. This workflow is important: machine learning need not be placed literally on the replacement critical path to influence hardware. It can expose predictive features, history lengths, and nonlinear interactions that are then distilled into conventional structures.

Parrot formulates cache replacement as imitation learning from Belady's policy [17]. The model learns a direct policy conditioned on past accesses and substantially improves hit rate in the paper's CPU and web-search experiments. However, the computational and storage demands of the learned model make it better viewed, for processor LLCs, as a research oracle and a demonstration of learnability than as a drop-in controller. Together, Glider and Parrot sharpen two questions: how much of optimal behavior is predictable from the past, and how can that knowledge be compressed into nanosecond-scale hardware?

D. Mockingjay: Multiclass Time-of-Reuse Mimicry

Mockingjay argues that binary cache-friendly/cache-averse prediction discards the ordering information required to mimic MIN faithfully [18]. It trains a PC-indexed reuse-distance predictor (RDP) using coarse timestamps from a sampled cache. Separate signatures distinguish accesses by the same PC depending on whether the access hit or missed. Temporal-difference-style updates reduce sensitivity to oscillating reuse distances and outliers [19]. The predicted reuse distance is converted into an estimated time of arrival (ETA) for a future reuse.

The LLC stores a compact estimated-time-remaining (ETR) counter per line rather than a full timestamp. ETR is initialized from the predicted reuse distance and aged as the set is accessed. Counters continue below zero when the predicted reuse time has passed. At eviction, Mockingjay chooses the line with the largest absolute ETR: either a line predicted far in the future or a line whose expected reuse is far overdue. Ties favor evicting a negative-ETR line. An incoming line whose predicted ETA is worse than all residents can be bypassed. Prediction is also refreshed on hits, so promotion is interpreted as a new ETA rather than a fixed recency jump [18].

In the paper's no-prefetch 100-mix multicore experiment, Mockingjay improved weighted speedup by 15.2% over LRU, compared with 7.6% for SHiP and 12.9% for Hawkeye. In its single-core evaluation, it achieved 5.7% improvement, close to the 6.0% improvement of the unrealizable MIN reference. For high-MPKI Championship Value Prediction workloads with a prefetcher, the reported improvement was 20.1% over LRU versus 13.4% for Hawkeye [18]. These results support the mechanism's central claim: multiclass reuse timing preserves more of MIN's ordering than binary labels.

10.48047/jocaaa.2024.33.06.201

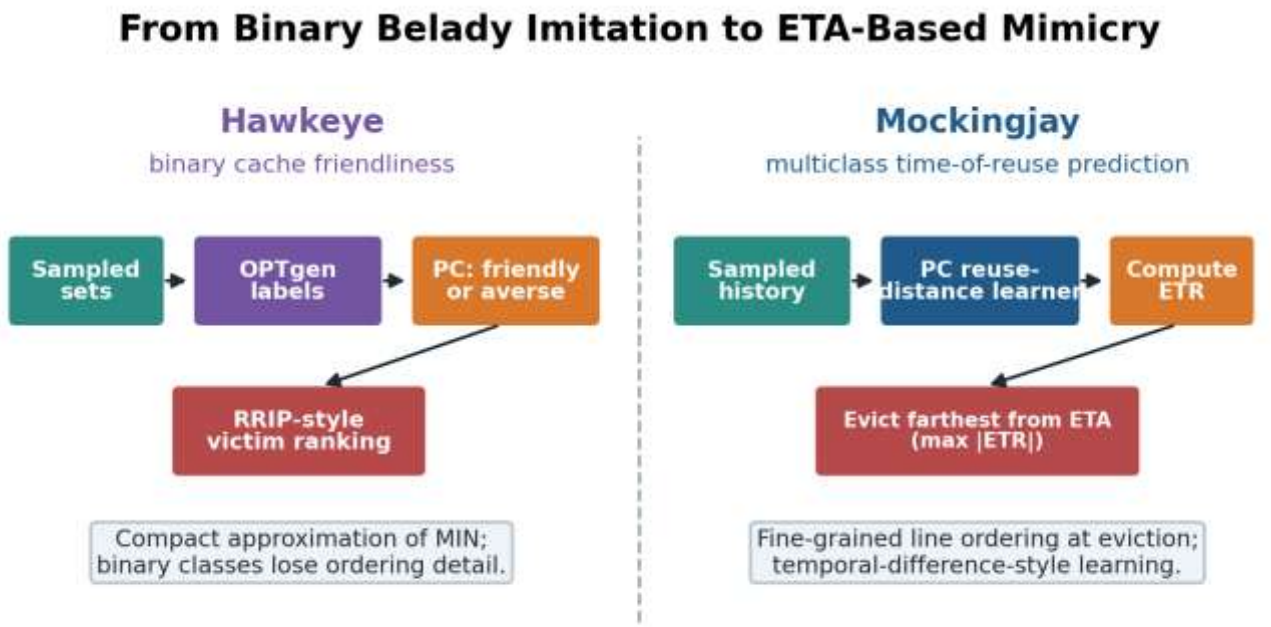

Fig. 6. Hawkeye learns a binary cache-friendliness label from OPTgen, whereas Mockingjay predicts reuse distance, maintains signed ETR state, and evicts the line farthest from its expected reuse time.

VII. COMPARATIVE ANALYSIS

A. Four Generations of Increasing Prediction Richness

The literature can be read as four successive relaxations of LRU. **First**, recency policies assume one ordering is universally useful. Their advantage is low design risk, deterministic behavior, and small latency. **Second**, adaptive insertion policies preserve the recency machinery but learn when an incoming line should not receive MRU protection. **Third**, reuse and dead-block predictors learn whether a context is likely to justify residency. **Fourth**, Belady-inspired policies learn cache utility or a finer time-of-reuse order. Each generation adds information while creating new state, training, and verification costs.

The progression is not monotonic for every workload. A stable loop may favor simple LRU or SRRIP because sophisticated predictors add little information. A pure stream favors BIP, DRRIP, bypass, or a dead-block predictor. A mixed application in which different PCs have stable behaviors favors SHiP or Hawkeye. Workloads with variable generations benefit from Leeway or multiperspective features. Fine-grained ETA prediction is most valuable when several resident lines are all broadly reusable but their reuse times differ enough that binary classification cannot rank them.

B. Multicore Awareness and Fairness

Only some policies are explicitly thread-aware. TADIP maintains per-thread insertion choices [6], and partitioning policies such as UCP and PIPP allocate capacity using per-application utility [4], [7]. Many reuse predictors mix training from all cores unless the core ID is incorporated into a signature. This can be beneficial when applications share behavior, but it can also create destructive aliasing or let a dominant thread control predictor state. Per-core predictors reduce interference but replicate storage and learn more slowly; shared predictors economize state but require careful hashing, tagging, or confidence control.

Fairness also changes the correct victim. Throughput-oriented replacement may repeatedly

10.48047/jocaaa.2024.33.06.201

sacrifice a low-IPC application's lines because doing so produces more aggregate instructions. A QoS-oriented controller may instead protect the application with the highest slowdown. One promising architecture is to separate policy layers: a QoS allocator establishes per-class occupancy or eviction eligibility, and a reuse predictor ranks lines within that domain. Another is to incorporate slowdown or criticality into the predictor's cost function, but doing so complicates training because the label is no longer simply hit or dead.

C. Metadata, Latency, and Energy

Metadata comparisons must include more than bit count. Exact LRU performs frequent ordering updates. RRIP adds a few bits per line but simple saturating operations. SHiP adds a signature history counter table and per-line outcome state. SDBP, Hawkeye, Leeway, perceptron, MPP, and Mockingjay add sampled histories or feature tables. The placement of these structures matters: a predictor accessed only on fill may be pipelined, while a predictor required before tag resolution can lengthen the hit path. Aging all lines eagerly is expensive; practical policies use coarse counters, sampled updates, or implicit epochs.

Energy has three components: predictor accesses, LLC metadata writes, and changed off-chip traffic. A policy with higher metadata energy can still reduce system energy if it prevents DRAM accesses. Conversely, a policy that improves hit rate but increases useless prefetch traffic may lose energy. The most credible evaluations report predictor area, access latency, dynamic and leakage energy, and sensitivity to realistic banking and port contention rather than only an abstract storage percentage.

TABLE II: REPRESENTATIVE RESULTS FROM ORIGINAL PAPERS (NOT CROSS-PAPER COMPARABLE)

Policy	Original evaluation context	Representative result reported by the paper
DIP [5]	1-core study; 1-MB, 16-way L2	21% average MPKI reduction vs. LRU; about two-thirds of LRU-to-OPT gap closed
TADIP [6]	2/4/8/16-core workloads	14%/18%/15%/17% average throughput gains over LRU
RRIP [8]	Single- and 4-core studies	SRRIP/DRRIP reported +4%/+10% single-core and +7%/+9% four-core throughput vs. LRU
SDBP [9]	Single-thread and multicore	11.7%/23% miss reduction; 5.9%/12.5% performance gain in reported configurations
SHiP [10]	Sequential and multiprogrammed	About 10% and 11%-12% performance gain over LRU, depending on configuration
Hawkeye [11]	Single-core and 4-core mixes	8.4% single-core and 15.0% four-core improvement over LRU
Perceptron [12]	Multiprogrammed study	3.2% false-positive rate reported; weighted-speedup gains over LRU
Leeway [13]	SPEC and CloudSuite	8.1% SPEC geomean for static Leeway; 5.4% CloudSuite for dynamic Leeway
MPP [14]	Single-thread and multiprogrammed	9.0% single-thread and 8.3% normalized weighted speedup over LRU
Mockingjay [18]	100 multicore mixes, no prefetch	15.2% over LRU; SHiP 7.6%, Hawkeye 12.9% in same study

10.48047/jocaaa.2024.33.06.201

Note: Values use the workload, simulator, cache, prefetch, and metric definitions of each cited paper. They are included to characterize evidence, not to rank policies across publications.

D. Practical Selection Guide

No single policy dominates all design points. A conservative commercial design prioritizing validation and short hit latency may select PLRU or SRRIP, possibly with a small set-dueling selector. A throughput-oriented server LLC with stable PC behavior may justify SHiP or Hawkeye. A research design targeting irregular, memory-intensive workloads may explore MPP or Mockingjay, provided the sampled structures and predictor accesses fit the cycle and energy budget. An architecture with strict isolation should combine replacement with partitioning and avoid predictor aliasing across security domains.

TABLE III: WORKLOAD-TO-POLICY DESIGN GUIDANCE

Workload / objective	Promising starting point	Reason
Streaming / scans	BIP, DRRIP, SHiP, SDBP	Do not grant one-touch fills high priority; bypass when confidence is strong
Cyclic working set > associativity	RRIP / DRRIP	Ordinal re-reference state breaks repeated LRU thrashing
Stable PC-correlated reuse	SHiP or Hawkeye	PC signatures generalize behavior with moderate hardware
Variable generations	Leeway, perceptron, MPP	Use distance tolerance or multiple features rather than one binary signature
Fine differences among reusable lines	Mockingjay	Predicted reuse distance provides an eviction order beyond live/dead
Aggressive data prefetching	Harmony or prefetch-aware Mockingjay	Evaluate demand misses and traffic together
QoS / hostile co-runners	Partitioning + thread-aware predictor	Constrain eligibility, then rank within the protected allocation

E. Common Failure Modes and Design Safeguards

Sophisticated replacement fails in recognizable ways. *Aliasing* merges unrelated signatures and trains a predictor toward an average that fits neither stream. *Staleness* preserves a once-correct prediction after a phase transition. *Sampling bias* teaches only the sets or PCs that happen to appear in the sampled structure. *Overconfidence* converts a noisy prediction into aggressive bypass or premature eviction. *Training interference* lets a high-rate core dominate shared predictor entries. These failures often explain why a policy with strong geomean performance has severe outliers.

Several safeguards recur across successful designs. Prediction counters should saturate and decay rather than retain unlimited history. Confidence should control the strength of the action: low-confidence lines can fall back to SRRIP instead of being bypassed. Core ID or protection-domain bits can be included in a signature when cross-thread aliasing is harmful. Multiple hash banks reduce systematic collisions. Set sampling should rotate or be checked for representativeness. Finally, a watchdog can compare the learned policy with a robust baseline in leader sets and disable prediction when it becomes harmful, extending DIP's set-dueling philosophy to modern predictors.

Error asymmetry deserves explicit design. A false prediction that a line is dead may destroy a critical future hit, whereas a false prediction that it is live merely consumes capacity. This

10.48047/jocaaa.2024.33.06.201

asymmetry argues for conservative bypass thresholds and for separating *admission* confidence from *victim-ranking* confidence. Mockingjay's signed ETR treatment is an example of graceful handling: an underestimated reuse time does not make a line permanently protected after its ETR reaches zero; the counter continues negative and eventually makes the line a candidate [18]. Leeway similarly changes orientation when generations are variable [13].

Validation should include adversarial microbenchmarks in addition to application suites. Streams reveal pollution resistance; cyclic traces reveal thrashing; alternating phases reveal adaptation delay; two PCs mapped to one predictor entry reveal alias sensitivity; imbalanced multicore mixes reveal training domination. These tests do not replace full-system workloads, but they make a policy's causal behavior understandable and help distinguish predictor bugs from workload-specific limitations.

VIII. OPEN RESEARCH CHALLENGES

A. QoS-Aware Prediction Rather Than QoS-Aware Allocation Alone

Most multicore policies optimize hits or throughput and add fairness through a separate allocator. Future predictors could learn the *system cost* of evicting a line, including the owning core's slowdown, memory-level parallelism, row-buffer locality, and service class. The challenge is creating stable labels: cost depends on future overlap and on how replacement changes progress. Hardware also needs guardrails so a noisy high-priority application cannot permanently capture capacity.

B. Prefetching, Coherence, and Writebacks

Replacement research often models clean demand lines, but real LLCs contain prefetched, dirty, shared, and coherence-active blocks. Evicting a dirty line incurs a writeback; evicting a shared line may trigger directory or snoop activity; retaining a prefetched line may save demand latency but consume bandwidth. A unified predictor should estimate multiple future events: demand reuse, prefetch re-arrival, writeback cost, coherence reuse, and pollution. Harmony [15] addresses a major part of this problem, while Mockingjay incorporates prefetch-aware intervals [18], but a general multicore cost model remains open.

C. Heterogeneous Cores, Sliced LLCs, and Chiplets

A line's value depends on which core will reuse it. High-performance and efficiency cores have different miss penalties and instruction throughput; accelerators may generate bursty, highly parallel traffic. Sliced LLCs add nonuniform access latency, and chiplets make remote-cache and memory costs topology-dependent. A replacement policy may need to predict both reuse time and reuse *location*. Training only on global set accesses can misorder lines when cores progress at different rates or when one slice becomes congested.

D. Security and Predictability

Deterministic replacement is observable through timing and can support eviction-based side channels. Randomization complicates attacks but can lower hit rate or interact unpredictably with learned state. Predictor tables themselves create shared microarchitectural state: one process may poison another's signature entries or infer behavior from replacement outcomes. Secure designs need domain-aware indexing, flush or partition rules, bounded training influence, and analysis of whether the policy amplifies covert channels. Accuracy, fairness, and security should be evaluated jointly rather than sequentially.

E. Robust Training and Fast Phase Adaptation

Sampled predictors must learn rapidly without overreacting. Aliasing, sparse observations, and delayed labels can leave stale state after a phase change. Confidence, decay, per-core tagging, temporal-difference updates, and multiple timescales are promising, but each adds state. A useful future direction is uncertainty-aware replacement: protect a line only when

10.48047/jocaaa.2024.33.06.201

prediction confidence justifies the opportunity cost, and fall back to a robust RRIP-like baseline when confidence collapses.

F. Reproducible and Representative Evaluation

Policy conclusions are sensitive to trace selection, warm-up, sampled-region length, cache size per core, prefetcher, memory latency, and the construction of multicore mixes. Researchers should release simulator code and policy parameters, report individual-workload distributions rather than only geomeans, include multiple cache capacities and core counts, and test emerging graph, analytics, cloud, and accelerator workloads in addition to SPEC. Results should include fairness, traffic, energy, and worst cases. At minimum, evaluations should include LRU or PLRU, SRRIP or DRRIP, SHiP, and one Belady-inspired policy so that conclusions do not depend on a weak reference point. A common evaluation framework would make it easier to distinguish genuine mechanism advances from favorable configuration choices.

TABLE IV: RECOMMENDED EVALUATION MATRIX FOR A NEW MULTICORE REPLACEMENT POLICY

Evaluation dimension	Minimum study	Question answered
Cache capacity / associativity	At least two capacities per core and two associativities	Separates mechanism benefit from simple overprovisioning
Core count and mixes	1-core plus multiple 2/4/8-core classes	Exposes interference, predictor sharing, and scalability
Workload diversity	SPEC, graph, cloud/data serving, and streaming microbenchmarks	Tests stable locality, irregular reuse, phases, and scans
Prefetch environment	No prefetch plus at least one modern prefetcher	Measures pollution, timeliness, and demand/traffic trade-offs
Memory system	Latency and bandwidth sensitivity; row-buffer statistics	Determines whether saved misses are performance-critical
Metrics	MPKI, IPC/weighted speedup, fairness, traffic, energy	Avoids optimizing hit rate while worsening system cost

G. From Predictor Accuracy to Implementation Closure

A replacement proposal is not complete when its predictor reaches high classification accuracy. It must close timing, storage, bandwidth, and verification constraints. Predictor lookup should be mapped onto the cache pipeline: the design must state whether the decision is needed before tag lookup, after miss detection, or only when a fill returns. Structures on the miss path can often tolerate several cycles, but state used on every hit must be banked and ported for the LLC access rate. A realistic design should also explain what happens when two accesses update the same predictor entry concurrently.

Storage accounting should distinguish per-line bits, per-set counters, sampled tags, signature tables, replacement queues, and error-correction overhead. A percentage of LLC data capacity can look small while still representing tens of kilobytes of multiported SRAM. Likewise, an aging rule that appears to be one counter decrement in pseudocode may require touching every way in a set. Practical designs use lazy aging, epochs, compressed timestamps, or arithmetic performed only during victim search. Mockingjay's coarse signed ETR counters and sampled timestamp history illustrate this kind of representational compression [18].

Verification is unusually important for adaptive replacement because state depends on long

10.48047/jocaaa.2024.33.06.201

histories and rare saturation cases. Designers should specify reset behavior, counter overflow, invalid-line priority, tie breaking, context switches, predictor flushing, and the interaction with way disabling or cache partitioning. Formal assertions can check invariants such as always selecting an eligible victim, never preferring an invalid line for eviction, and maintaining bounded counter state. Differential trace testing against a simple reference model can validate policy updates independently of the processor simulator.

Finally, implementation closure should include a fallback mode. If predictor SRAM is unavailable because of a fault, power gating, or an unsupported cache configuration, the LLC should degrade to SRRIP or PLRU without losing correctness. A fallback also supports online monitoring: leader sets can estimate whether the learned policy is currently helping, and the controller can reduce prediction strength when the benefit disappears. This approach converts a replacement predictor from a monolithic heuristic into a supervised optimization layer above a robust baseline.

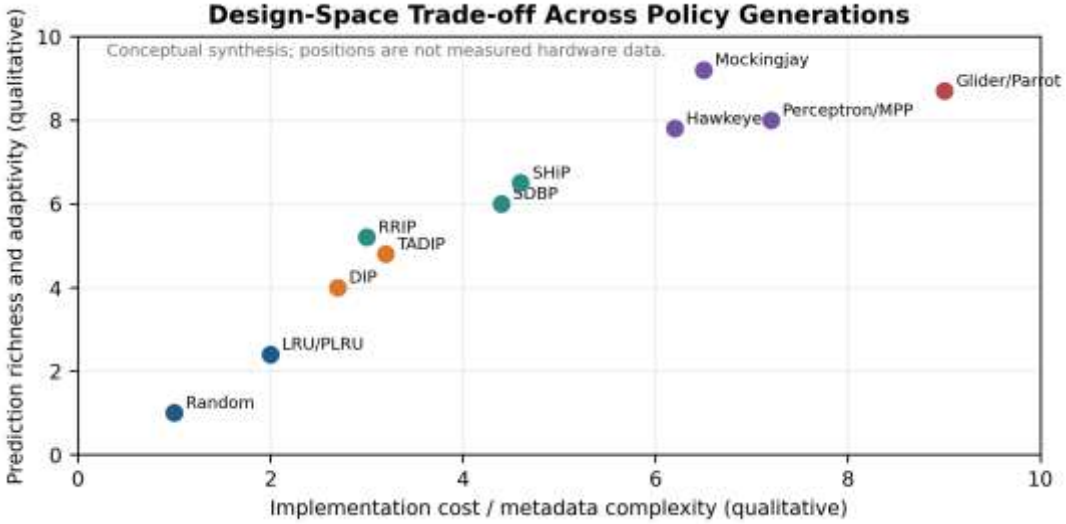

Fig. 7. Qualitative design-space synthesis. Coordinates are explanatory rather than measured hardware data: prediction richness generally grows with metadata, training, and implementation complexity.

IX. DESIGN PRINCIPLES FOR FUTURE POLICIES

A. Separate Admission, Retention, and Eviction

A cache controller should not force one score to answer three different questions. Admission asks whether an incoming line deserves any residency; retention asks whether a hit or elapsed time should change its protection; eviction asks which of the current residents has the lowest future value. The evidence and error costs differ. A conservative admission stage can prevent scan pollution, a reuse predictor can update retention after observed hits, and a fine-grained distance or cost estimator can order only the small set of victim candidates. This modularity also permits a simple policy to remain available when a predictor has low confidence.

B. Match Adaptation Scope to the Interference Domain

Global state is economical but can hide heterogeneity. Per-core state captures distinct applications but duplicates storage and may fail when threads migrate. Per-PC state follows code but aliases data structures and security domains. Per-set state reflects local pressure but learns slowly. The appropriate granularity should follow the source of nonstationarity: per-thread selectors for heterogeneous co-runners, per-signature reuse tables for stable producers, per-set aging for local contention, and domain tags where isolation matters. Hierarchical

10.48047/jocaaa.2024.33.06.201

designs can share coarse statistics while retaining small per-core corrections.

C. Couple Confidence to Action Strength

Prediction confidence should determine how disruptive an action may be. High-confidence deadness can justify bypass; medium confidence can justify distant insertion; low confidence should fall back to SRRIP or PLRU. The same principle applies to promotion: one hit need not erase all prior evidence, and a predicted reuse distance can be refreshed without moving a line unconditionally to the highest priority. Confidence-aware control reduces catastrophic false positives and makes learned policies easier to deploy because the baseline bounds behavior when training is sparse or stale.

D. Predict System Cost, Not Only the Next Reference

The final objective is not reuse prediction accuracy. It is lower execution time, energy, and interference subject to fairness and security constraints. A future policy should distinguish a critical demand reuse from an overlapped reuse, account for dirty-writeback and coherence cost, recognize whether a prefetch will recreate the line, and include topology-dependent latency in sliced or chiplet caches. The predictor may remain approximate, but its label should align with the system objective. This shift from access prediction to cost prediction is the natural next step after Mockingjay's finer ordering of reuse time.

X. CONCLUSION

Cache replacement has evolved from maintaining a recency order to learning a prediction of future value. LRU remains a vital baseline because it is understandable, local, and effective for stable locality, but it fails when insertion grants protection to scans or when multicore streams interfere. DIP and TADIP taught the cache to choose an insertion mode online. RRIP replaced exact ordering with compact re-reference classes. SDBP and SHiP used sampled histories and signatures to distinguish useful from dead or low-reuse blocks. Perceptron, MPP, and Leeway enriched the context and modeled variability. Hawkeye used Belady's policy as a teacher, Harmony incorporated prefetch interactions, and Glider and Parrot demonstrated how learning can expose more powerful predictors. Mockingjay completed the requested lineage by estimating reuse distance and using signed ETR state to approximate a fine-grained future-use ordering.

The historical trajectory does not eliminate simpler policies. It clarifies when complexity is justified. The best practical controller is the simplest one that predicts the workload property responsible for misses, adapts at the correct thread and phase granularity, and fits the metadata, latency, energy, fairness, and security budget of the target system. Future progress will likely come from cost-aware prediction that unifies reuse with QoS, prefetching, coherence, topology, and uncertainty rather than from hit-rate prediction alone.

TABLE V: PRACTICAL DEPLOYMENT CHECKLIST FOR A REPLACEMENT POLICY

Deployment question	Evidence required before adoption
Workload evidence	Measure stream intensity, reuse-distance spread, phase duration, co-runner sensitivity, and prefetch coverage.
Decision scope	State explicitly whether the policy controls admission, retention, promotion, bypass, victim selection, or several stages.
Hardware closure	Account for all per-line, sampled, and predictor state; bank/port the structures; report access timing and update frequency.
Adaptation safety	Define confidence, decay or reset behavior, alias control, leader-set monitoring,

10.48047/jocaaa.2024.33.06.201

Deployment question	Evidence required before adoption
	and a robust SRRIP/PLRU fallback.
System interactions	Evaluate dirty victims, coherence, prefetching, QoS/partitioning, thread migration, and protection-domain isolation.
Validation evidence	Use adversarial microbenchmarks and diverse multicore mixes; report throughput, fairness, traffic, energy, outliers, and sensitivity.

REFERENCES

[1] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," IBM Systems Journal, vol. 5, no. 2, pp. 78–101, 1966.
[2] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation techniques for storage hierarchies," IBM Systems Journal, vol. 9, no. 2, pp. 78–117, 1970.
[3] A. J. Smith, "Cache memories," ACM Computing Surveys, vol. 14, no. 3, pp. 473–530, Sep. 1982.
[4] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in Proc. 39th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO), 2006, pp. 423–432, doi: 10.1109/MICRO.2006.49.
[5] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive insertion policies for high performance caching," in Proc. 34th Annu. Int. Symp. Computer Architecture (ISCA), 2007, pp. 381–391, doi: 10.1145/1250662.1250709.
[6] A. Jaleel, W. Hasenplaugh, M. K. Qureshi, J. Sebot, S. C. Steely, and J. Emer, "Adaptive insertion policies for managing shared caches," in Proc. 17th Int. Conf. Parallel Architectures and Compilation Techniques (PACT), 2008, pp. 208–219, doi: 10.1145/1454115.1454145.
[7] Y. Xie and G. H. Loh, "PIPP: Promotion/insertion pseudo-partitioning of multi-core shared caches," in Proc. 36th Annu. Int. Symp. Computer Architecture (ISCA), 2009, pp. 174–183, doi: 10.1145/1555754.1555778.
[8] A. Jaleel, K. B. Theobald, S. C. Steely, Jr., and J. Emer, "High performance cache replacement using re-reference interval prediction (RRIP)," in Proc. 37th Annu. Int. Symp. Computer Architecture (ISCA), 2010, pp. 60–71, doi: 10.1145/1815961.1815971.
[9] S. M. Khan, Y. Tian, and D. A. Jiménez, "Sampling dead block prediction for last-level caches," in Proc. 43rd Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO), 2010, pp. 175–186, doi: 10.1109/MICRO.2010.24.
[10] 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. 44th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO), 2011, pp. 430–441.
[11] A. Jain and C. Lin, "Back to the future: Leveraging Belady's algorithm for improved cache replacement," in Proc. 43rd Annu. Int. Symp. Computer Architecture (ISCA), 2016, pp. 78–89, doi: 10.1109/ISCA.2016.17.
[12] E. Teran, Z. Wang, and D. A. Jiménez, "Perceptron learning for reuse prediction," in Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO), 2016, pp. 1–12, doi: 10.1109/MICRO.2016.7783705.
[13] P. Faldu and B. Grot, "Leeway: Addressing variability in dead-block prediction for last-level caches," in Proc. 26th Int. Conf. Parallel Architectures and Compilation Techniques (PACT), 2017, pp. 180–193.

10.48047/jocaaa.2024.33.06.201

[14] D. A. Jiménez and E. Teran, "Multiperspective reuse prediction," in Proc. 50th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO), 2017, pp. 436–448, doi: 10.1145/3123939.3123942.
[15] A. Jain and C. Lin, "Rethinking Belady's algorithm to accommodate prefetching," in Proc. 45th Annu. Int. Symp. Computer Architecture (ISCA), 2018, pp. 110–123, doi: 10.1109/ISCA.2018.00020.
[16] Z. Shi, X. Huang, A. Jain, and C. Lin, "Applying deep learning to the cache replacement problem," in Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO), 2019, pp. 413–425, doi: 10.1145/3352460.3358319.
[17] E. Z. Liu, M. Hashemi, K. Swersky, P. Ranganathan, and J. Ahn, "An imitation learning approach for cache replacement," in Proc. 37th Int. Conf. Machine Learning (ICML), PMLR, vol. 119, 2020, pp. 6237–6247.
[18] I. Shah, A. Jain, and C. Lin, "Effective mimicry of Belady's MIN policy," in Proc. IEEE Int. Symp. High-Performance Computer Architecture (HPCA), 2022, pp. 558–572, doi: 10.1109/HPCA53966.2022.00048.
[19] R. S. Sutton, "Learning to predict by the methods of temporal differences," Machine Learning, vol. 3, pp. 9–44, 1988, doi: 10.1007/BF00115009.