International Journal of Applied Mathematics
Volume 38 No. 4, 2025
ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)

PREFETCHING TECHNIQUES IN MODERN MULTICORE PROCESSORS: PRINCIPLES, TAXONOMY, DESIGN TRADE-OFFS, AND EMERGING DIRECTIONS

[1]Purnendu Das, [2,*] Bishwa Ranjan Roy

[1,2]Department of Computer Science, Assam University, Silchar.

Abstract

Prefetching is one of the most effective microarchitectural techniques for tolerating the widening processor–memory latency gap. Its role becomes more complex in multicore processors because each core may generate speculative memory traffic that competes for shared cache capacity, on-chip-network bandwidth, memory-controller queues, and DRAM service. This paper reviews the evolution of prefetching from software hints, next-line and stride mechanisms to history-, delta-, spatial-, temporal-, perceptron-, and reinforcement-learning-based designs. It first explains the organization of modern multicore cache hierarchies and the latency-hiding objective of prefetching. It then develops a unified design taxonomy based on trigger source, context, prediction representation, lookahead, target cache, filtering, and feedback control. Representative techniques—including stream buffers, GHB, SMS, AMPM, VLDP, BOP, SPP, Domino, Bingo, PPF, MLOP, DSPatch, IPCP, Pythia, and Berti—are compared qualitatively. Particular attention is given to multicore interference, bandwidth-aware throttling, prefetch-aware cache and memory management, security leakage, and evaluation methodology. The review concludes that future prefetchers should be multilevel, resource-aware, workload-adaptive, and explicitly secure rather than optimized only for single-core address-prediction accuracy.

Index Terms—data prefetching, multicore processors, cache memory, spatial locality, temporal correlation, hardware prefetcher, memory latency, reinforcement learning.

I. INTRODUCTION

The latency of off-chip memory has not scaled at the rate of core execution throughput, instruction-window capacity, or cache bandwidth. This long-standing “memory wall” means that a single last-level-cache miss can consume hundreds of processor cycles, particularly when bank conflicts, queuing, coherence actions, or memory-side contention are present [2]. Larger caches and wider out-of-order windows help, but they cannot eliminate compulsory misses or irregular working sets. Prefetching attacks the problem from a different direction: it predicts future instruction or data blocks and initiates their transfer before a demand access reaches the miss point.

A prefetch is valuable only when three conditions hold. First, the requested line must later be consumed; otherwise the transfer wastes energy and bandwidth. Second, the line must arrive early enough to overlap most of the demand miss latency. Third, it must not evict more valuable demand data or delay requests from another core. These conditions create an inherently multi-objective problem. A conservative prefetcher can be accurate but cover few misses, whereas an aggressive prefetcher can increase coverage while polluting caches and

International Journal of Applied Mathematics
Volume 38 No. 4, 2025
ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)

saturating memory queues. The same design may help a bandwidth-rich desktop workload but harm a four- or eight-core mix that shares the LLC and DRAM channels.

Research has therefore evolved from simple sequential prediction to context-rich learning and system feedback. Stream buffers [3], stride predictors [5], Markov correlation [6], and GHB structures [7] established the foundations. Spatial and temporal designs such as SMS [8], AMPM [12], and STeMS [13] widened the range of detectable patterns. More recent mechanisms use multistep delta histories, confidence-controlled lookahead, multiple spatial representations, filtering, classification, and online learning, exemplified by VLDP [17], BOP [18], SPP [19], Bingo [23], PPF [24], DSPatch [26], IPCP [27], Pythia [28], and Berti [30].

This review is organized around the architectural questions that remain stable across implementations: what event triggers learning, what context identifies a stream, how future addresses are represented, where prefetched lines are placed, and how aggressiveness is controlled under shared-resource pressure. The emphasis is on modern multicore processors, although software and instruction prefetching are included to show how the broader latency-tolerance design space fits together.

II. MULTICORE CACHE HIERARCHY AND THE PURPOSE OF PREFETCHING

A. Private and Shared Levels

A contemporary multicore processor typically combines private L1 instruction and data caches, a private or semi-private L2, a sliced shared LLC, a coherent on-chip interconnect, and one or more memory controllers (Fig. 1). Private levels provide low hit latency and isolate common working sets. The LLC captures inter-core sharing, reduces off-chip traffic, and mediates capacity among cores. Because each level has different latency, bandwidth, visibility, and pollution sensitivity, prefetch placement is a first-order design decision. An L1 prefetcher observes detailed load-PC information and can deliver very low hit latency, but its prediction must be computed quickly and its mistakes consume scarce capacity. An L2 or LLC prefetcher can use more history and storage, but it has less time to hide latency and often sees a filtered request stream.

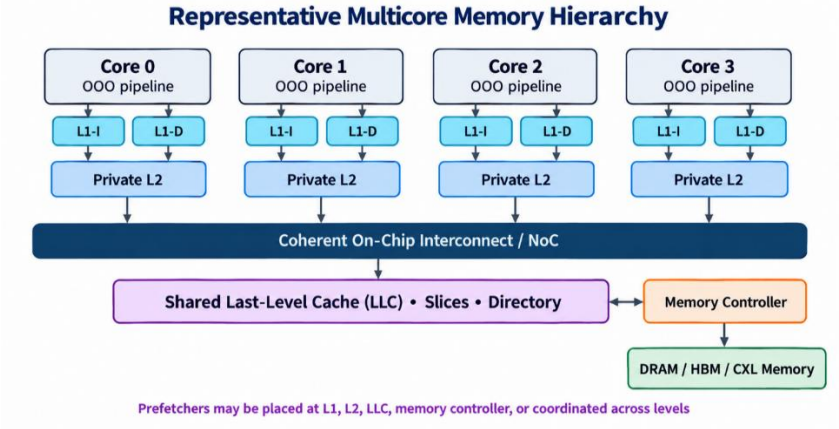

Fig. 1. Representative multicore processor and memory hierarchy. Prefetching can be deployed at private-cache, shared-cache, and memory-controller levels.

International Journal of Applied Mathematics
Volume 38 No. 4, 2025
ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)

B. Cache Organization and Placement

A set-associative cache divides an address into tag, set index, and block offset. Demand or prefetch requests compare their tags with all ways in the selected set and, on a miss, allocate a miss-status entry and eventually choose a victim. Prefetch metadata is usually maintained outside the normal tag/data arrays, although training can use hits, misses, evictions, or fills. Figure 2 shows a generic organization in which an access detector updates history, a predictor creates candidate addresses, a confidence filter decides which candidates deserve issue, and a request queue enforces resource limits.

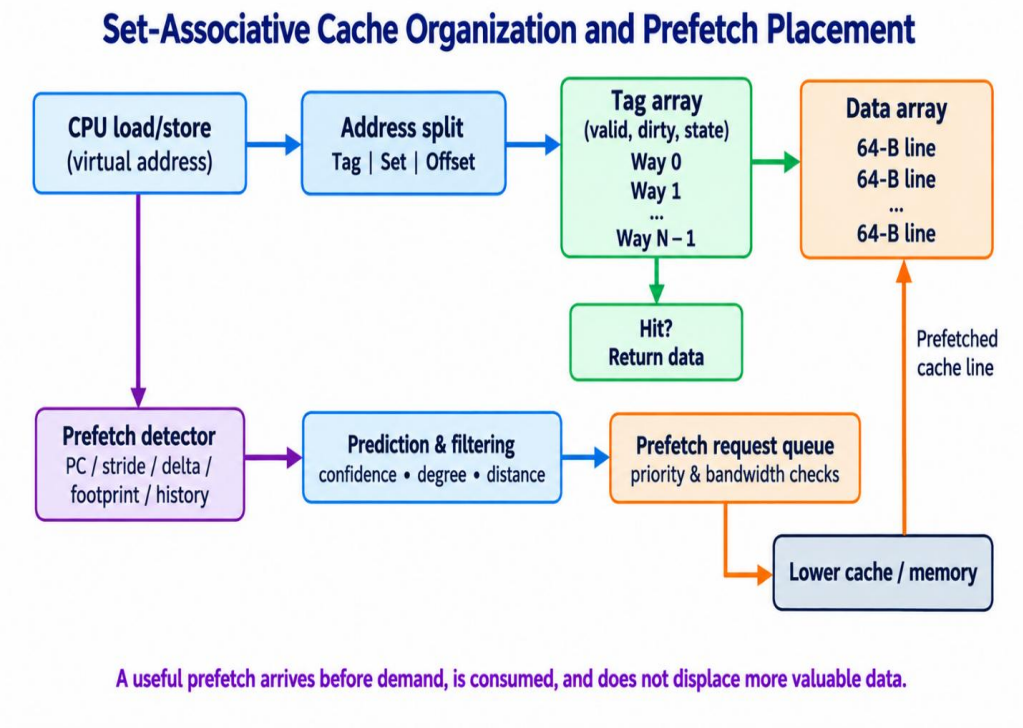

Fig. 2. Set-associative cache organization and common prefetcher placement. Prediction, filtering, and request scheduling are logically separate functions.

C. Latency Hiding, Not Merely Address Prediction

The main purpose of prefetching is to convert exposed stall time into overlapped memory activity. If a demand miss would incur latency Lm and a prefetch starts Δ cycles earlier, the residual exposed latency is approximately max(0, Lm − Δ), assuming that the prefetch does not suffer additional interference. This simplified expression highlights why an accurate but late prefetch can have little performance value. Conversely, a very distant prefetch may arrive early but occupy the cache too long and displace useful data. Timeliness is therefore coupled to prefetch distance, program progress, memory-level parallelism, and current queueing delay.

International Journal of Applied Mathematics
Volume 38 No. 4, 2025
ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)

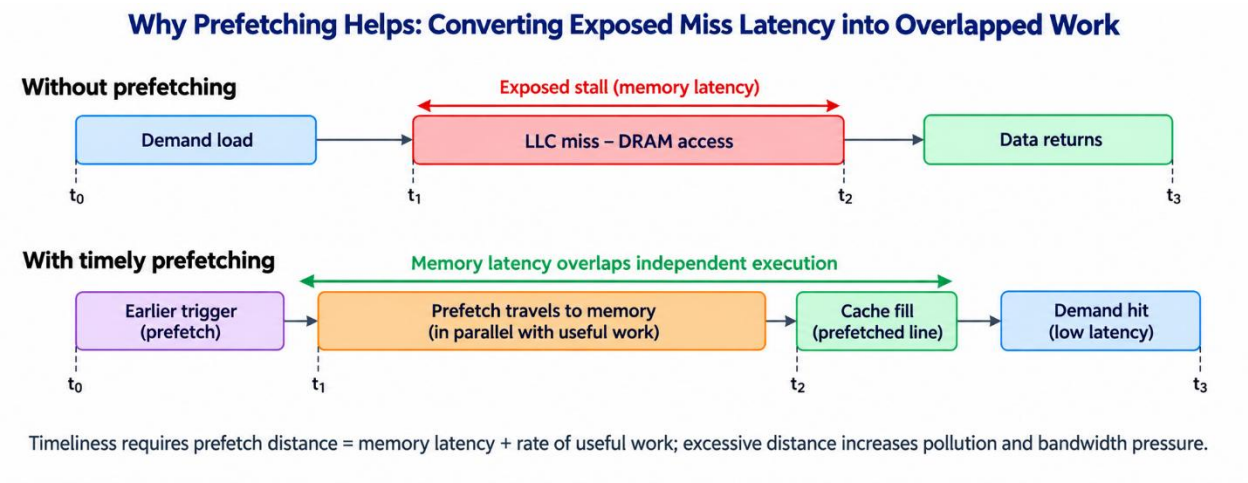

Fig. 3. A timely prefetch overlaps the long-latency transfer with useful execution and converts a future miss into a cache hit.

Software prefetching exposes the desired lookahead in instructions inserted by the compiler or programmer. Mowry, Lam, and Gupta showed that loop analysis can identify array references and select a prefetch distance that balances latency and overhead [4]. Hardware prefetching instead infers patterns dynamically and requires no source changes, making it more robust to binaries and phase changes. Instruction prefetching targets front-end stalls, while data prefetching targets load and store misses. Execution-based techniques such as runahead or helper threads execute selected instructions ahead of retirement and indirectly generate memory requests; they are powerful for pointer-dependent patterns but consume core resources.

III. A UNIFIED PREFETCHER DESIGN SPACE

Although prefetchers are often named after a particular prediction algorithm, a complete design is better described as a pipeline of observation, learning, candidate generation, filtering, placement, and feedback (Fig. 4). Separating these functions helps explain why two designs with similar predictors can behave differently in a multicore system.

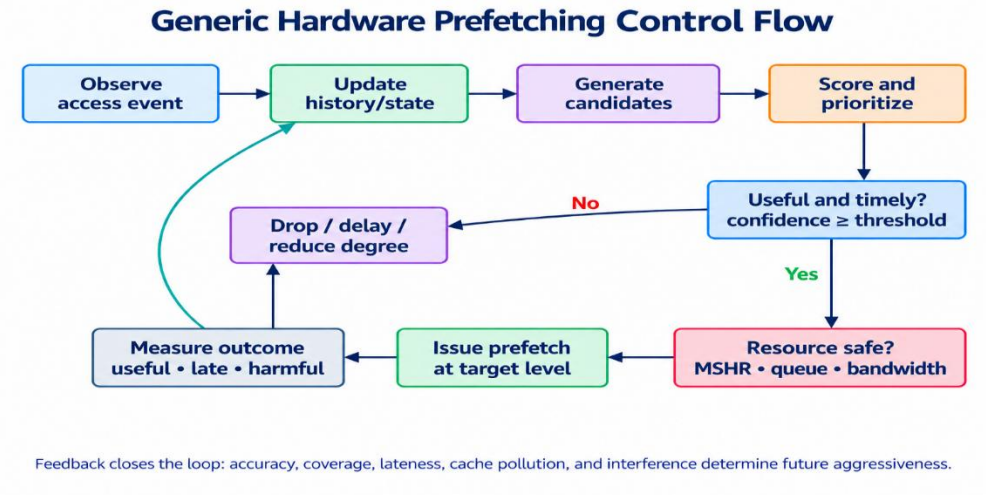

Fig. 4. Generic hardware prefetching control flow with confidence and resource checks.

International Journal of Applied Mathematics
Volume 38 No. 4, 2025
ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)

A. Trigger and Context

The trigger can be every load, an L1 miss, an L2 miss, a cache fill, or a region transition. Training on all accesses reveals more regularity but increases port pressure and metadata traffic. Miss-only training focuses on costly references but hides intervening hits. Context may be global, per-PC, per-page, per-region, per-core, or a hash of several features. PC context distinguishes unrelated streams generated by different instructions; page context constrains arithmetic to legal physical pages; region context enables compact bitmaps; and global context can detect phase-level temporal sequences.

B. Prediction Representation

Sequential and stride schemes predict an arithmetic relation between addresses. Delta schemes model a sequence of differences, often within a page. Spatial schemes record which offsets in a region tend to be used after a trigger. Temporal schemes replay recurring address orders. Learned schemes treat candidate selection or filtering as classification or reinforcement learning. These categories overlap: SPP stores compressed path signatures built from deltas, Bingo associates spatial footprints with short and long events, and IPCP classifies PCs before applying specialized sub-prefetchers.

C. Degree, Distance, and Destination

Degree is the number of lines issued per trigger; distance is how far ahead prediction reaches. Higher degree improves coverage for dense streams but increases traffic. The destination can be L1, L2, LLC, or a dedicated buffer. A line predicted with high confidence may be placed near the core, while a lower-confidence line can be staged farther away. Multilevel designs must also prevent duplicate requests and account for whether a lower-level prefetcher sees demand accesses, upper-level prefetches, or both.

D. Quality Metrics

Accuracy is the fraction of issued prefetches that are later used; coverage is the fraction of baseline misses eliminated; timeliness measures whether useful prefetches arrive before demand; and pollution captures harmful displacement. These metrics are not interchangeable. Accuracy can increase when degree is reduced even as performance falls, and coverage can increase while weighted speedup drops because prefetch traffic delays critical demand requests. A rigorous evaluation therefore reports system performance and resource cost together.

TABLE I: CORE METRICS FOR PREFETCHER EVALUATION

Metric	Definition / observation	Desired trend	Common pitfall
Accuracy	Useful prefetches ÷ issued prefetches	Higher	Can favor timid designs
Coverage	Demand misses eliminated ÷ baseline	Higher	May ignore extra traffic

International Journal of Applied Mathematics
Volume 38 No. 4, 2025
ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)

Metric	Definition / observation	Desired trend	Common pitfall
	demand misses		
Timeliness	Useful lines present before demand	Higher	Late "useful" lines still stall
Pollution	Demand lines displaced by speculative fills	Lower	Hard to attribute after cascaded evictions
Bandwidth overhead	Extra cache/memory transfers	Lower	Aggregate traffic hides burst pressure
Speedup / weighted speedup	Performance relative to no-prefetch baseline	Higher	Single-core IPC misses fairness

E. End-to-End Request Lifecycle and Failure Modes

A candidate address becomes useful only after surviving several architectural checks. The trigger first identifies a context worth consulting, after which the prediction engine proposes one or more lines. Address validation prevents illegal page crossings and translates virtual information when necessary. Duplicate filters then remove lines already present in a cache, miss-status entry, writeback buffer, or another prefetch queue. Finally, the scheduler decides whether sufficient cache, network, and memory resources exist and chooses a target level. Figure 5 summarizes this lifecycle and highlights that each stage can independently determine performance.

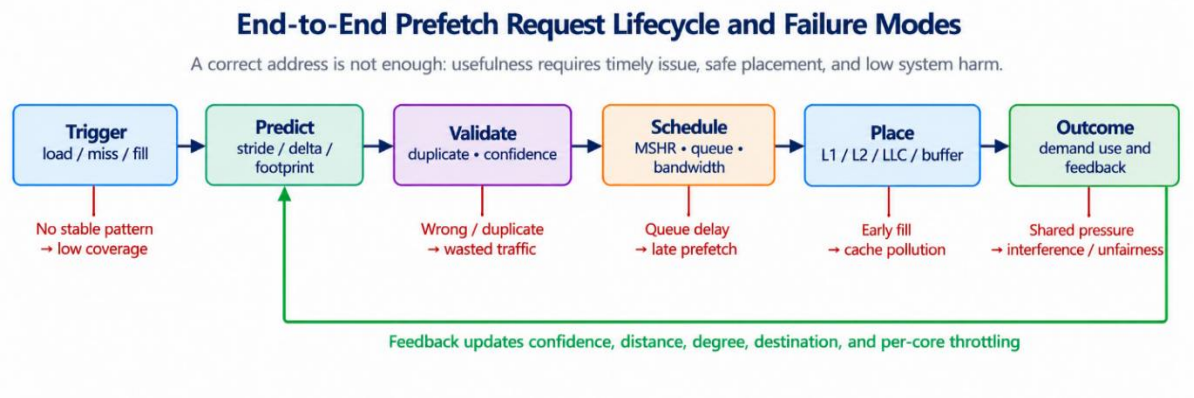

Fig. 5. End-to-end lifecycle of a prefetch request and the principal ways in which an otherwise plausible prediction can become harmful.

International Journal of Applied Mathematics
Volume 38 No. 4, 2025
ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)

The failure modes explain why address-prediction accuracy is an incomplete objective. A pattern can be correct but too sparse to cover meaningful misses. A candidate can duplicate an in-flight request, arrive after the demand, or arrive so early that it is evicted before use. Even a timely line can be harmful if its fill displaces critical data or if its request delays another core. Attribution is therefore important: hardware should distinguish inaccurate, late, redundant, polluted, and interference-causing requests rather than counting every unused line as the same event.

Feedback should act on the stage responsible for the loss. Low pattern confidence calls for suppressing or retraining the generator; persistent lateness calls for greater distance or higher scheduling priority; early eviction calls for a different destination or insertion policy; and global queue pressure calls for lower degree or per-core throttling. This staged diagnosis is more stable than a single accuracy threshold because it avoids fixing one problem by creating another. It also provides a natural interface between specialized per-core predictors and a shared multicore controller.

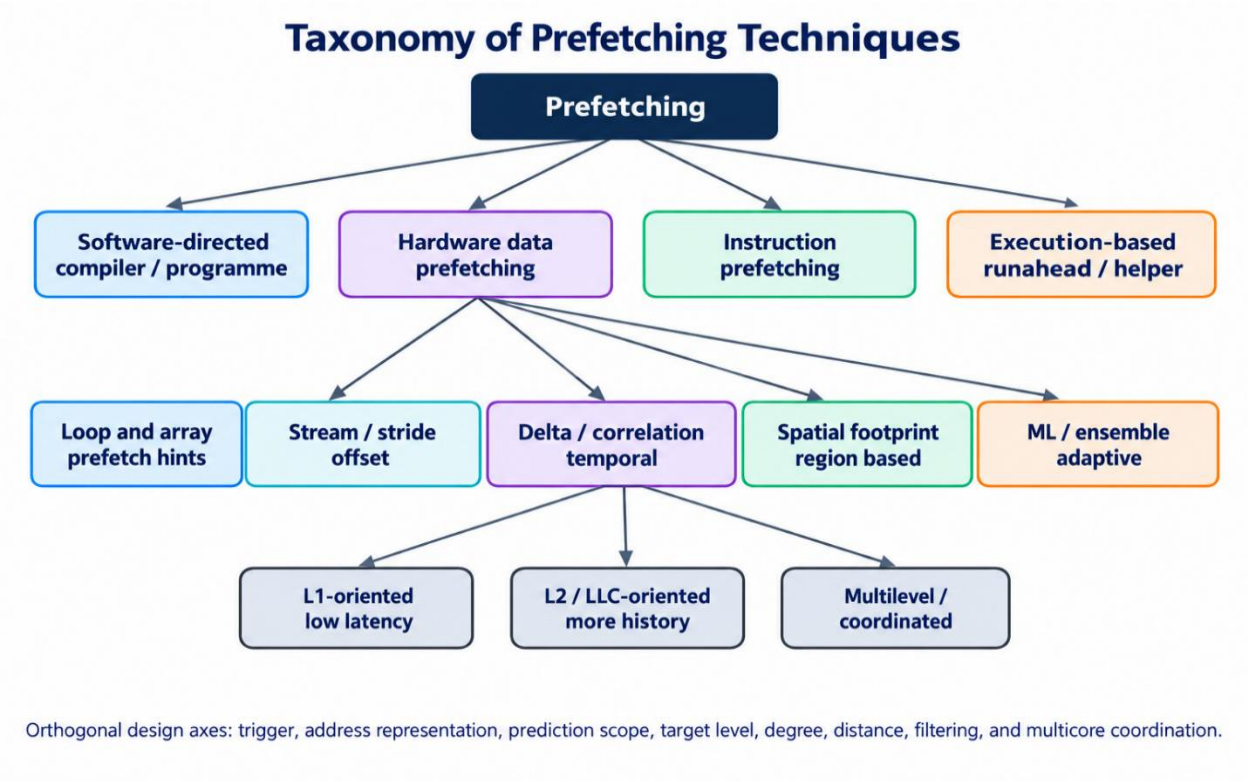

Fig. 6. Taxonomy of prefetching techniques and orthogonal implementation choices.

IV. CLASSICAL PREFETCHING TECHNIQUES

A. Software-Directed Prefetching

Software prefetch instructions are attractive when the compiler can prove a stable relationship between loop iterations and memory addresses. The compiler estimates reuse, cache-line granularity, and a distance measured in iterations, then inserts nonbinding requests that should not raise architectural exceptions [4]. The main advantages are semantic visibility and

Volume 38 No. 4, 2025
ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)

minimal predictor hardware. The disadvantages are instruction overhead, register pressure, dependence on a specific cache and memory latency, and difficulty with irregular data structures. Profile-guided approaches can reduce these limitations by identifying hot prefetch kernels and selecting distances offline, an idea revisited in recent temporal work [36].

B. Sequential, Stream, and Stride Prefetching

Next-line prefetching assumes that block X+1 will follow block X. Jouppi's stream buffers separate prefetched blocks from the demand cache and extend a stream after a miss [3]. Multiple stream buffers tolerate interleaved sequences, while stride predictors associate a PC with a constant difference between addresses [5]. These mechanisms are compact, fast, and effective for arrays and scans. Their weaknesses are overfetching at stream boundaries, inability to follow changing deltas, and aggressive traffic when several cores scan large regions concurrently. Adaptive stream detection modulates the number and depth of streams using observed usefulness [9].

C. Correlation and History-Based Prefetching

Markov prefetching predicts the next miss address from previously observed transitions [6]. It handles nonuniform sequences but can require large tables and generate several successors. GHB reorganizes recent miss addresses into a global FIFO linked to index entries, allowing stride or address-correlation algorithms to share compact history [7]. Temporal schemes record and replay miss sequences that recur across phases. Their potential coverage is high for stable server or graph phases, but metadata can be large and replay may become inaccurate when interleaving changes.

D. Spatial and Region-Based Prefetching

Spatial prefetchers use the observation that an instruction often touches a repeatable subset of cache lines within a page or region. SMS associates a trigger PC and region offset with a footprint of subsequent accesses [8]. AMPM maintains access maps and detects parallel offset relationships inside a page [12]. STeMS combines temporal ordering of regions with intra-region spatial patterns [13]. Region bitmaps compactly describe many candidates, but fixed boundaries can fragment patterns and the first access to a new region is often difficult to predict.

International Journal of Applied Mathematics
Volume 38 No. 4, 2025
ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)

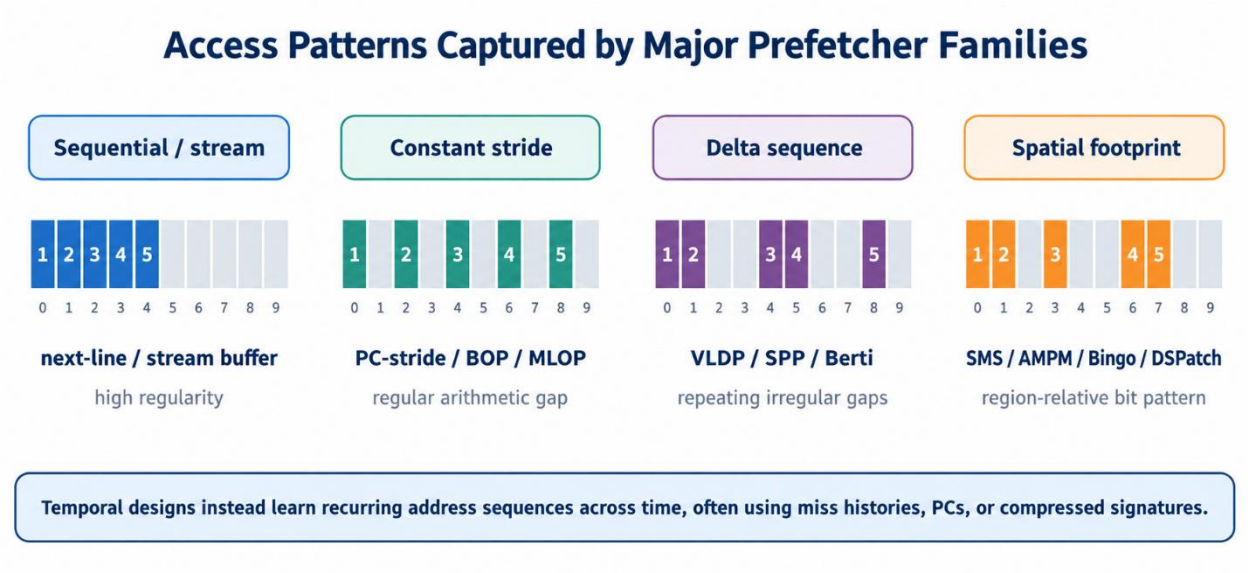

Fig. 7. Simplified examples of access patterns targeted by major hardware-prefetcher families.

E. Instruction and Execution-Based Prefetching

Instruction streams are generally more predictable than data because control flow constrains the next fetch addresses. Temporal Instruction Fetch Streaming records recurring L1 instruction-miss sequences and replays them from lower cache levels [11]. Modern front ends also prefetch branch-target structures and code lines using profile or history information. Execution-based mechanisms such as runahead continue speculative execution after a long-latency miss and convert future independent loads into prefetches. They can discover pointer-derived addresses unavailable to conventional stride logic, but they consume rename, scheduling, and execution energy and require careful suppression of low-value instructions.

TABLE II: QUALITATIVE COMPARISON OF CLASSICAL PREFETCHER FAMILIES

Family	Representative work	Metadata / context	Best suited to	Main limitation
Sequential/stream	Stream buffers [3]	Miss address, stream state	Dense scans	Boundary overfetch
PC-stride	Stride directed [5]	PC, last address, stride	Regular loops	Changing/interleaved strides
Correlation	Markov [6], GHB [7]	Address transition history	Recurring irregular sequences	Table size and aliasing
Spatial footprint	SMS [8], AMPM [12]	PC/page/region bitmap	Stable layout within pages	First miss and region boundaries
Temporal	STeMS [13],	Recurring miss	Phase-repetitive	History storage

International Journal of Applied Mathematics
Volume 38 No. 4, 2025
ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)

Family	Representative work	Metadata / context	Best suited to	Main limitation
	Domino [20]	sequences	workloads	and drift
Execution based	Runahead/helper execution	Dynamic dependence chain	Pointer-derived accesses	Core energy and complexity

V. REPRESENTATIVE MODERN DATA PREFETCHERS

A. Delta and Offset Predictors

VLDP learns variable-length sequences of deltas within a physical page and consults prediction tables from longer to shorter histories [17]. It is more expressive than a constant-stride predictor while retaining page-local arithmetic. BOP chooses the offset that best balances prediction and timeliness during repeated test rounds [18]. MLOP extends offset selection by learning multiple lookahead levels, improving coverage when one offset is insufficient [25]. SPP hashes recent deltas into a signature, uses the signature to predict a distribution of next deltas, and recursively follows high-confidence paths [19]. Its confidence-based lookahead is influential because it decouples pattern learning from aggressiveness.

B. Temporal and Spatial Predictors

Domino improves temporal prediction by linking an address with additional contextual history, reducing ambiguity when one address has several successors [20]. Bingo observes that short events such as a PC generalize well while longer events such as PC plus address context provide precision. It stores both associations in a unified structure and selects the available pattern with the most appropriate confidence [23]. DSPatch maintains two modulated spatial patterns: one coverage-oriented and one accuracy-oriented. Runtime bandwidth utilization determines which pattern is used, making resource pressure part of prediction rather than an afterthought [26].

C. Filtering, Classification, and Learning

PPF attaches a perceptron filter to an underlying lookahead predictor. Features describing the request and system state are combined into a score that decides whether the candidate should be issued or placed at a particular cache level [24]. IPCP classifies load PCs into streaming, constant-stride, complex-stride, or irregular behavior and applies a small specialized mechanism to each class [27]. This “division by behavior” avoids forcing one representation onto every instruction. Pythia formulates the decision as online reinforcement learning: program context defines a state, candidate deltas are actions, and rewards capture usefulness together with bandwidth cost [28]. Berti places a local-delta predictor at L1D, uses PC-local timing information to learn which prior access could have generated a timely prefetch, and orchestrates fills across levels [30].

International Journal of Applied Mathematics
Volume 38 No. 4, 2025
ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)

Recent work continues to refine these directions. Matryoshka coalesces delta sequences to reduce redundant representation [29]. Load-criticality-aware prefetching prioritizes requests that are more likely to block execution when bandwidth is constrained [31]. Page-and-PC delta prediction combines local page behavior with instruction context [34], while complexity-effective local-delta designs reduce metadata and lookup cost without abandoning accurate PC-local learning [35]. These designs show that modern progress often comes from better attribution, filtering, and resource control rather than from a completely new address representation.

TABLE III. REPRESENTATIVE MODERN HARDWARE DATA PREFETCHERS

Techniq ue	Yea r	Core idea	Typical level	Strength	Trade-off
VLDP [17]	201 5	Variable-length page deltas	L2	Complex repeating deltas	Several history tables
BOP [18]	201 6	Select best timely offset	L2/LLC	Simple and timing aware	Limited irregular coverage
SPP [19]	201 6	Signature + confidence lookahead	L2	Multistep delta paths	Aliasing and traffic
Domino [20]	201 8	Context-rich temporal successor	L2/LLC	Disambiguates temporal links	History footprint
Bingo [23]	201 9	Short/long event spatial patterns	L2	High spatial coverage	Pattern-table cost
PPF [24]	201 9	Perceptron candidate filter	With SPP	Improves aggressiveness control	Feature/weight storage
DSPatch [26]	201 9	Accuracy/coverage spatial modes	L2	Bandwidth-adaptive	Region dependence
IPCP [27]	202 0	PC behavior classification	L1/L2	Versatile, compact sub-engines	Classifier stability
Pythia [28]	202 1	Online RL with system reward	L1/L2	Holistic adaptation	Training and action design
Berti [30]	202 2	Timely PC-local delta learning	L1D	Strong local attribution	L1 metadata/latenc y

International Journal of Applied Mathematics
Volume 38 No. 4, 2025
ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)

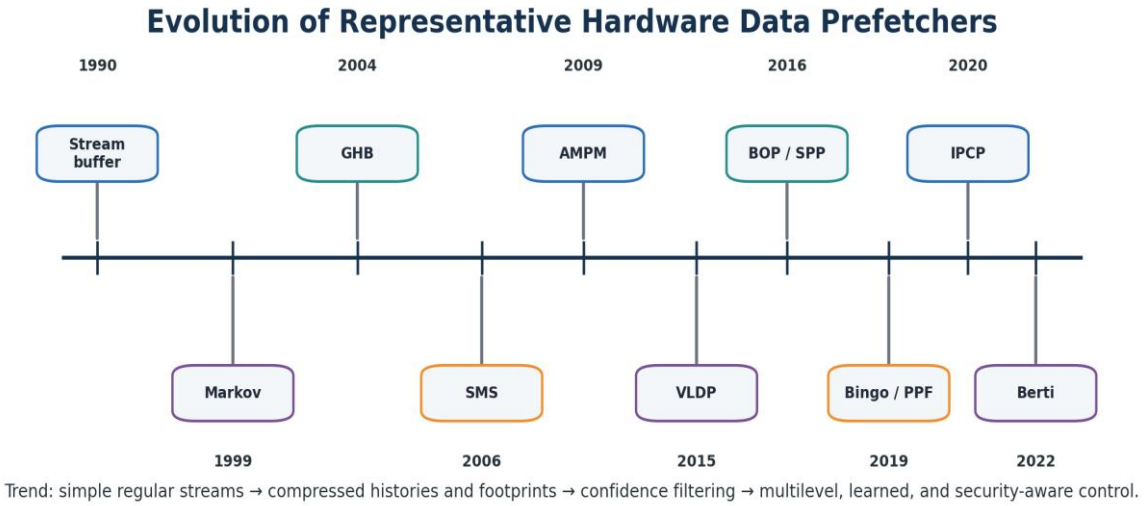

Fig. 8. Evolution from simple streams to history-, footprint-, filter-, and learning-based prefetching.

D. Selecting and Composing Prefetchers

The preferred mechanism depends on where the dominant predictability appears. Dense unit-stride scans rarely justify a large learned predictor: a stream or offset engine can provide high coverage with little metadata. Stable non-unit-stride loops favor PC-local stride or delta learning. Sparse but repeatable accesses within a page benefit from a spatial footprint, whereas pointer-linked structures require temporal correlation, address-generation assistance, or execution-based lookahead. The most difficult workloads interleave several behaviors under the same core; for them, classification and candidate filtering are often more valuable than simply increasing history depth.

Placement must match the predictor's confidence and reaction time. An L1 engine sees the richest instruction context and has the least time to decide; it should therefore use fast, compact tables and issue only highly confident requests. L2 engines receive a more selective stream and can afford deeper lookahead. LLC or memory-side engines have greater visibility across cores and pages, but they cannot easily recover the originating instruction and may deliver requests too late for short-latency misses. A useful hierarchy assigns near-core predictors to attribution and global predictors to long-range correlation, while a shared duplicate filter prevents several levels from fetching the same line.

Hybrid designs should also divide responsibility rather than allowing all engines to run at maximum degree. A PC classifier can route regular loads to a stride engine, region-repetitive loads to a footprint engine, and uncertain candidates to a learned filter. The controller can then choose destination and priority according to confidence, expected reuse distance, criticality, and bandwidth pressure. This modular organization improves explainability: each issued request has an identifiable generator, confidence source, and resource decision. It also permits selective disabling when a component becomes inaccurate or a security-domain transition requires state isolation.

International Journal of Applied Mathematics
Volume 38 No. 4, 2025
ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)

TABLE IV: PRACTICAL MAPPING FROM ACCESS BEHAVIOR TO PREFETCHER ORGANIZATION

Observed behavior	Suitable mechanism	Useful context	Preferred action under pressure
Dense sequential scan	Next-line, stream, or best-offset	Recent misses; page boundary	Reduce degree before disabling
Stable non-unit stride	PC-stride or local delta	Load PC and last deltas	Preserve critical streams
Repeated region footprint	SMS, AMPM, Bingo, DSPatch	PC/page/region bitmap	Use accuracy-oriented footprint
Recurring irregular order	GHB, Markov, Domino, temporal replay	Address sequence and history	Shorten replay or demote fills
Interleaved load behaviors	IPCP-style classification	PC class and phase state	Disable low-utility class only
Ambiguous candidates	PPF or compact learned ranker	Confidence and system features	Raise issue threshold
Highly dynamic workload	Feedback/RL controller	Usefulness, lateness, pressure	Optimize global utility budget

No mapping is universal because address regularity and system pressure vary over time. Consequently, modern prefetchers should be judged by how quickly they identify a phase, how much traffic is spent during learning, and how safely they back off after a wrong classification. A design that wins only after a long warmup can be ineffective for short server requests, while a rapidly adapting prefetcher may overreact to noise. The key architectural lesson is that prediction, filtering, placement, and throttling should expose separate control points even when they are implemented in one compact hardware structure.

VI. PREFETCHING IN MULTICORE PROCESSORS

A. Why Single-Core Optimization Is Insufficient

In a single core, an inaccurate prefetch mainly wastes that core’s cache capacity and memory bandwidth. In a multicore processor, the same request can delay another thread at every shared structure: MSHRs, LLC banks, network links, memory-controller queues, DRAM banks, and writeback buffers. A high-coverage prefetcher can therefore improve its local IPC while reducing system throughput or fairness. The problem becomes acute when one bandwidth-intensive core generates easy sequential streams and another core executes latency-critical, low-memory-level-parallelism code.

International Journal of Applied Mathematics
Volume 38 No. 4, 2025
ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)

B. Feedback and Coordinated Control

Feedback Directed Prefetching monitors accuracy, lateness, and pollution and adjusts aggressiveness rather than using a fixed degree [10]. Coordinated control extends the idea across cores by considering interference and global resource pressure, dynamically selecting each core's prefetch setting [14]. The controller can disable a low-utility prefetcher, lower its degree, reduce distance, demote fills to LLC, or deprioritize speculative memory commands. The important distinction is that control decisions optimize system-level benefit rather than the apparent quality of each prefetcher in isolation.

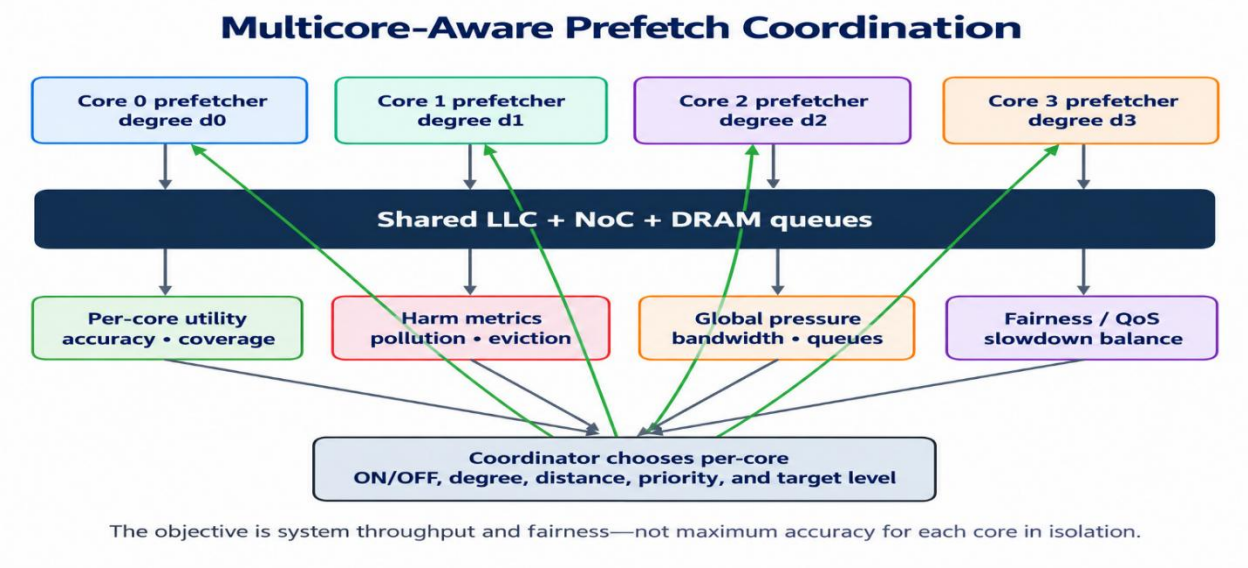

Fig. 9. Multicore-aware coordination uses per-core utility and global harm signals to choose aggressiveness, priority, and destination.

C. Fairness, QoS, and Phase Behavior

Throughput metrics such as weighted speedup should be paired with harmonic speedup or maximum slowdown because prefetching can redistribute service unevenly. A fair controller may reserve bandwidth for demand requests, cap per-core speculative occupancy, or throttle the core whose prefetches create the highest marginal interference. Phase changes complicate control: short windows react quickly but are noisy; long windows are stable but may preserve a harmful setting after a phase transition. Hybrid controllers often combine fast pressure signals with slower utility estimates.

D. Shared Cache and Memory Scheduling

Prefetch-aware cache management can preferentially evict unused prefetched lines, protect demand lines, or place speculative lines with low replacement priority. Prefetch-aware memory controllers can prioritize demand requests, exploit row-buffer locality, and drop redundant or excessively late prefetches. However, strict demand priority may starve useful prefetches and reduce future parallelism. The best policy is criticality aware: it gives demand requests precedence under pressure but permits timely speculative requests when queue headroom exists.

International Journal of Applied Mathematics
Volume 38 No. 4, 2025
ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)

TABLE V: MULTICORE PREFETCHING PROBLEMS AND CONTROL RESPONSES

Problem	Observable signal	Possible action	Risk if overused
LLC pollution	Unused prefetch evictions, demand victimization	Low insertion priority; bypass; selective fill	Useful prefetched lines become late
Bandwidth saturation	Channel utilization, queue occupancy	Reduce degree/distance; disable core	Coverage collapse after short bursts
Cross-core unfairness	Per-core slowdown or stall cycles	Per-core caps; demand reservation	Lower total throughput
Duplicate multilevel requests	Same block in multiple prefetch queues	Cross-level filter and ownership	Extra lookup/control latency
Late prefetches	Demand merges with in-flight prefetch	Increase distance or prioritize request	More early pollution
Phase instability	Abrupt accuracy/MPKI change	Reset confidence; shorten adaptation window	Oscillation

VII. INTERACTION WITH CACHE, COHERENCE, AND MEMORY SUBSYSTEMS

A. Replacement and Insertion

Prefetching changes the meaning of recency. A newly filled line is not necessarily valuable, and a prefetched line that has not yet been demanded may need a grace period. Inserting every prefetch at the most-recently-used position can evict demand data, whereas inserting all prefetches near the least-recently-used position can discard accurate but early lines. Effective designs distinguish demand reuse from speculative age and may train replacement only when a prefetch is actually consumed. Prefetch-aware variants of reuse-prediction and dead-block policies are therefore natural companions to aggressive predictors.

B. Coherence and Shared Data

A read prefetch can trigger coherence lookups and consume directory bandwidth even when it never reaches the core. In producer–consumer workloads, a prefetch may arrive before the producer writes the final value, causing an invalidation and a second transfer. Coherence-aware prefetching should avoid ownership requests for speculative reads, recognize shared or migratory patterns, and suppress lines with repeated invalidation before use. At the same time, temporal or dependence-based mechanisms can exploit producer patterns to reduce communication latency when the sharing relation is stable.

International Journal of Applied Mathematics
Volume 38 No. 4, 2025
ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)

C. Address Translation and Page Boundaries

Many data prefetchers operate on physical addresses after translation, which prevents them from learning across virtual pages and adds TLB latency to training. L1 prefetchers may use virtual offsets but must avoid crossing into an unrelated physical page. Page-local prediction simplifies correctness but misses large contiguous structures. Cross-page designs need translation lookahead, page-contiguity confidence, or operating-system support. The same issue appears in CXL and heterogeneous-memory systems, where a predicted address may map to a tier with very different latency and bandwidth.

D. DRAM and Heterogeneous Memory

Prefetch streams can either improve or destroy DRAM row-buffer locality. Sequential requests often cluster within a row, whereas aggressive interleaving from many cores can increase bank conflicts. HBM provides more bandwidth but does not eliminate queueing or cache pollution, so designs such as DSPatch explicitly scale aggressiveness with available bandwidth [26]. In tiered DRAM–NVM or CXL systems, the prefetch destination and migration policy should be coordinated: moving a speculative line to a scarce fast tier may be more costly than serving it remotely on demand.

VIII. EVALUATION METHODOLOGY FOR A FAIR REVIEW

Prefetchers should be evaluated with diverse access patterns, including SPEC CPU, server and cloud workloads, graph analytics, databases, and multiprogrammed mixes. Reporting only memory-intensive benchmarks exaggerates potential benefit, while averaging every workload equally can hide regressions on already cache-resident programs. A useful methodology groups workloads by baseline MPKI, memory-level parallelism, and available bandwidth, then reports both per-workload distributions and geometric means.

Simulation must clearly specify the core, cache sizes and latencies, replacement policy, MSHR counts, prefetch queue sizes, DRAM channels, warmup, and instruction count. The prefetcher's metadata and lookup latency should be modeled, not treated as an infinite zero-cycle table. For multicore mixes, weighted speedup, harmonic speedup, maximum slowdown, total DRAM traffic, row-buffer hit rate, LLC MPKI, and energy are more informative than IPC alone. Sensitivity to metadata budget, bandwidth scaling, cache size, and prefetch target level helps distinguish algorithmic merit from a favorable configuration.

Comparisons should use tuned, publicly available implementations where possible. The Data Prefetching Championships and ChampSim ecosystem have helped standardize traces and interfaces, but implementation details still matter: whether training uses hits or misses, whether prefetches train lower-level predictors, how duplicate requests are filtered, and whether a late prefetch is credited as useful. A review table should therefore compare not only headline speedup but also context, target level, storage, and control assumptions.

- Use a no-prefetch baseline and at least one simple production-like stream/stride baseline.
- Report prefetch accuracy, coverage, timeliness, pollution, bandwidth, and system performance together.

Volume 38 No. 4, 2025
ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)

• Include single-core and multicore configurations, plus bandwidth- and cache-capacity sensitivity.

• Account for metadata area, predictor access latency, dynamic energy, and additional cache/memory traffic.

• Disclose harmful outliers and phase behavior rather than relying only on a geometric mean.

IX. SECURITY, ENERGY, AND RELIABILITY

A. Prefetcher State as a Side Channel

Prefetchers learn from access history and leave observable changes in cache state, making predictor tables and generated lines potential side channels. Work on hidden hardware-prefetcher leakage showed that an attacker can monitor prefetch behavior to infer sensitive victim accesses [22]. AfterImage demonstrated control-flow and load tracking through an instruction-pointer stride prefetcher [32]. More recent attacks use prefetchers as speculative transmission mechanisms, showing that cache-only defenses can be insufficient [38]. These results are especially relevant to multicore systems because predictor state or downstream cache effects may be shared across security domains.

B. Security-Aware Design

Possible defenses include partitioning or flushing predictor state at domain switches, tagging entries with address-space identifiers, restricting training across privilege boundaries, suppressing cross-domain prefetch fills, and designing secure-cache-compatible prefetch ordering [33]. These measures carry performance and storage costs. A secure design should define what state is shared, when it is reset, and whether speculative requests can influence structures visible to another domain. Security evaluation should include both confidentiality leakage and denial-of-service through bandwidth amplification.

C. Energy and Reliability

Every unused prefetch consumes predictor energy, interconnect activity, cache writes, and possibly DRAM activation/precharge energy. High accuracy does not guarantee energy efficiency if the mechanism requires large associative tables. Prefetch-induced writes can also increase wear in NVM-based caches and memory. Energy-aware prefetching should include metadata lookup, extra fills, writebacks caused by pollution, and memory-controller activity. Reliability concerns include queue overflow, thermal concentration in LLC slices, and unfair bandwidth capture under sustained streaming.

X. EMERGING DIRECTIONS

A. Learning with Explicit Resource Budgets

Pythia illustrates the potential of reinforcement learning to combine program context with system feedback [28]. The next step is budget-constrained learning: an agent should maximize reduced stall time subject to explicit limits on bytes, cache occupancy, and inference latency. Small contextual bandits or table-based value functions are more plausible for near-term hardware than deep networks. Multi-agent formulations can coordinate per-core

Volume 38 No. 4, 2025
ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)

policies while sharing global pressure information, an area receiving renewed attention for multicore prefetching [37].

B. Criticality and Semantic Context

Not all misses matter equally. A load at the head of the reorder buffer with no independent work is more valuable than a miss overlapped by many others. CLIP and related designs incorporate load criticality into selection under bandwidth pressure [31]. Future predictors can combine PC/delta patterns with dependency depth, branch confidence, memory-level parallelism, and software semantics. Compiler or profile annotations can identify graph frontiers, sparse tensor phases, and pointer-chasing kernels without dictating exact addresses.

C. Cross-Layer and Heterogeneous Prefetching

Prefetching should increasingly be treated as a hierarchy-wide service rather than a collection of isolated per-cache engines. A cross-layer manager can assign responsibility for each pattern, choose a destination based on confidence and reuse distance, coordinate translation, and prevent duplicates. Heterogeneous cores, chiplets, CXL-attached memory, and near-memory accelerators make static placement less effective. A fast core may require deeper lookahead than an efficiency core, and remote memory may justify software-visible or profile-guided hints.

D. Robustness and Explainability

A practical prefetcher must degrade gracefully on unseen patterns, phase changes, adversarial accesses, and tight bandwidth. Confidence should therefore be calibrated, not merely ranked. Designers should expose counters that explain why requests were issued, dropped, or throttled. Such observability aids validation, security auditing, and operating-system control. The most promising direction is a hybrid architecture: simple specialized predictors generate candidates, a compact learned filter ranks them, and a multicore controller enforces global resource and security policies.

E. Deployment Checklist for Future Designs

A deployable prefetcher must satisfy constraints that are often secondary in algorithm-focused studies. Its metadata should fit within a clearly stated area budget, predictor lookup should not lengthen the cache hit path, and updates should avoid excessive ports or associative searches. Recovery behavior is equally important: confidence must decay after phase changes, stale state should be reclaimed, and request generation must remain bounded when the predictor encounters adversarial or random streams. Multilevel implementations also require an ownership rule so that an L1, L2, and LLC engine do not learn from and reproduce the same speculative request.

International Journal of Applied Mathematics
Volume 38 No. 4, 2025
ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)

TABLE VI. CHECKLIST FOR A PRACTICAL MULTICORE PREFETCHER

Design question	Recommended practice	Reason
What is optimized?	Use critical stall reduction and system utility, not accuracy alone	Connects prediction to performance and fairness
Where is a line placed?	Select L1/L2/LLC destination from confidence and expected reuse distance	Balances latency hiding against pollution
How is pressure handled?	Reserve demand resources and throttle per core using queue and bandwidth signals	Prevents speculative traffic from blocking demand
How is state isolated?	Tag, partition, or reset predictor state at protection boundaries	Reduces cross-domain leakage and mistraining
How is complexity reported?	Model bytes, ports, lookup latency, energy, and warmup behavior	Avoids zero-cost metadata assumptions
How are failures exposed?	Report harmful workloads, late requests, and phase-recovery time	Shows robustness rather than only average benefit

These requirements suggest a layered implementation strategy. Small deterministic engines should recognize common streams at low latency; specialized delta or footprint components should add coverage only for attributable patterns; a compact filter should rank ambiguous candidates; and a global controller should enforce bandwidth, fairness, energy, and security budgets. Such separation does not prevent learning, but it confines learning to decisions for which adaptation is valuable and leaves safety-critical resource limits explicit. This balance between specialization and coordination is likely to matter more than continually enlarging a monolithic history table.

XI. CONCLUSION

Prefetching remains essential because it attacks memory latency before a demand access stalls the core. The field has progressed from simple streams and constant strides to compressed delta histories, spatial footprints, temporal replay, behavior classification, perceptron filtering, and online reinforcement learning. Yet prediction sophistication alone does not determine value. Timeliness, placement, cache pollution, duplicate traffic, coherence effects, and shared-resource interference are equally important.

For modern multicore processors, the appropriate objective is system-level reduction in critical stall time under bandwidth, capacity, fairness, energy, and security constraints. This favors multilevel and multicore-aware designs that separate candidate generation from filtering and global control. Future prefetchers are likely to be hybrid: they will combine interpretable pattern engines with learned ranking, exploit load criticality and software

Volume 38 No. 4, 2025
ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)

semantics, adapt to heterogeneous memory, and isolate state across protection domains. Evaluation must reflect these goals through transparent metadata budgets, realistic queueing, multicore mixes, and explicit reporting of harmful cases.

REFERENCES

[1] S. Mittal, "A survey of recent prefetching techniques for processor caches," ACM Computing Surveys, vol. 49, no. 2, Art. 35, 2016, doi: 10.1145/2907071.

[2] W. A. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious," ACM SIGARCH Computer Architecture News, vol. 23, no. 1, pp. 20–24, 1995.

[3] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in Proc. 17th Int. Symp. Computer Architecture (ISCA), 1990, pp. 364–373.

[4] T. C. Mowry, M. S. Lam, and A. Gupta, "Design and evaluation of a compiler algorithm for prefetching," in Proc. 5th Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS), 1992, pp. 62–73.

[5] J. W. C. Fu, J. H. Patel, and B. L. Janssens, "Stride directed prefetching in scalar processors," in Proc. 25th Annu. Int. Symp. Microarchitecture (MICRO), 1992, pp. 102–110.

[6] D. Joseph and D. Grunwald, "Prefetching using Markov predictors," IEEE Trans. Computers, vol. 48, no. 2, pp. 121–133, 1999.

[7] K. J. Nesbit and J. E. Smith, "Data cache prefetching using a global history buffer," in Proc. 10th Int. Symp. High-Performance Computer Architecture (HPCA), 2004, pp. 96–105.

[8] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Spatial memory streaming," in Proc. 33rd Int. Symp. Computer Architecture (ISCA), 2006, pp. 252–263.

[9] I. Hur and C. Lin, "Memory prefetching using adaptive stream detection," in Proc. 39th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO), 2006, pp. 397–408.

[10] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt, "Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers," in Proc. 13th Int. Symp. High-Performance Computer Architecture (HPCA), 2007, pp. 63–74.

[11] M. Ferdman, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Temporal instruction fetch streaming," in Proc. 41st Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO), 2008, pp. 1–10.

[12] Y. Ishii, M. Inaba, and K. Hiraki, "Access map pattern matching for data cache prefetch," in Proc. Int. Conf. Supercomputing (ICS), 2009, pp. 499–500.

[13] S. Somogyi, T. F. Wenisch, A. Ailamaki, and B. Falsafi, "Spatio-temporal memory streaming," in Proc. 36th Int. Symp. Computer Architecture (ISCA), 2009, pp. 69–80.

[14] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt, "Coordinated control of multiple prefetchers in multi-core systems," in Proc. 42nd Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO), 2009, pp. 316–326.

[15] A. Jain and C. Lin, "Linearizing irregular memory accesses for improved correlated prefetching," in Proc. 46th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO), 2013, pp. 247–259.

Volume 38 No. 4, 2025
ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)

[16] S. H. Pugsley et al., "Sandbox prefetching: Safe run-time evaluation of aggressive prefetchers," in Proc. 20th Int. Symp. High-Performance Computer Architecture (HPCA), 2014, pp. 626–637.

[17] M. Shevgoor, S. Koladiya, R. Balasubramonian, C. Wilkerson, S. H. Pugsley, and Z. Chishti, "Efficiently prefetching complex address patterns," in Proc. 48th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO), 2015.

[18] P. Michaud, "Best-offset hardware prefetching," in Proc. IEEE Int. Symp. High Performance Computer Architecture (HPCA), 2016, pp. 469–480.

[19] J. Kim, S. H. Pugsley, P. V. Gratz, A. L. N. Reddy, C. Wilkerson, and Z. Chishti, "Path confidence based lookahead prefetching," in Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO), 2016, pp. 1–12.

[20] M. Bakhshalipour, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Domino temporal data prefetcher," in Proc. IEEE Int. Symp. High Performance Computer Architecture (HPCA), 2018, pp. 131–142.

[21] S. Kondguli and M. Huang, "Division of labor: A more effective approach to prefetching," in Proc. 45th Annu. Int. Symp. Computer Architecture (ISCA), 2018, pp. 83–95.

[22] Y. Shin, H. Kim, D. Kim, J. Jeong, and J. Hur, "Unveiling hardware-based data prefetcher, a hidden source of information leakage," in Proc. ACM SIGSAC Conf. Computer and Communications Security (CCS), 2018.

[23] M. Bakhshalipour, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Bingo spatial data prefetcher," in Proc. IEEE Int. Symp. High Performance Computer Architecture (HPCA), 2019, pp. 399–411.

[24] E. Bhatia, G. Chacon, S. H. Pugsley, E. Teran, P. V. Gratz, and D. A. Jiménez, "Perceptron-based prefetch filtering," in Proc. 46th Annu. Int. Symp. Computer Architecture (ISCA), 2019, doi: 10.1145/3307650.3322207.

[25] M. Shakerinava, M. Bakhshalipour, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Multi-lookahead offset prefetching," in Proc. 3rd Data Prefetching Championship (DPC3), 2019.

[26] R. Bera, A. V. Nori, O. Mutlu, and S. Subramoney, "DSPatch: Dual spatial pattern prefetcher," in Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO), 2019, pp. 531–544.

[27] S. Pakalapati and B. Panda, "Bouquet of instruction pointers: Instruction pointer classifier-based spatial hardware prefetching," in Proc. 47th Annu. Int. Symp. Computer Architecture (ISCA), 2020, doi: 10.1109/ISCA45697.2020.00021.

[28] R. Bera, K. Kanellopoulos, A. V. Nori, T. Shahroodi, S. Subramoney, and O. Mutlu, "Pythia: A customizable hardware prefetching framework using online reinforcement learning," in Proc. 54th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO), 2021.

[29] S. Jiang et al., "Matryoshka: A coalesced delta sequence prefetcher," in Proc. 30th Int. Conf. Parallel Architectures and Compilation Techniques (PACT), 2021, doi: 10.1145/3472456.3473510.

[30] A. Navarro-Torres et al., "Berti: An accurate local-delta data prefetcher," in Proc. 55th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO), 2022, pp. 975–991.

Volume 38 No. 4, 2025
ISSN: 1311-1728 (printed version); ISSN: 1314-8060 (on-line version)

[31] B. Panda et al., "CLIP: Load criticality based data prefetching for bandwidth-constrained systems," in Proc. 56th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO), 2023, doi: 10.1145/3613424.3614245.

[32] Y. Chen et al., "AfterImage: Leaking control flow data and tracking load operations via hardware prefetcher," in Proc. 28th ACM Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2023, doi: 10.1145/3575693.3575719.

[33] S. Nath et al., "Secure prefetching for secure cache systems," in Proc. 57th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO), 2024, IEEE Xplore document 10764627.

[34] Y. Cui et al., "A highly effective page and PC based delta prefetcher," ACM Trans. Architecture and Code Optimization, 2024, doi: 10.1145/3675398.

[35] A. Navarro-Torres et al., "A complexity-effective local delta prefetcher," IEEE Trans. Computers, 2025, doi: 10.1109/TC.2025.3533086.

[36] M. Li et al., "Profile-guided temporal prefetching," in Proc. 52nd Annu. Int. Symp. Computer Architecture (ISCA), 2025, doi: 10.1145/3695053.3731070.

[37] R. Bera et al., "Multi-agent reinforcement learning for multicore prefetching," in Proc. 58th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO), 2025, doi: 10.1145/3725843.3756096.

[38] F. Jiang et al., "SpectrePrefetch: Undermining cache-centric secure speculation defenses or secure cache designs via hardware prefetchers," 2025, IEEE Xplore document 11240638.