Exploiting Program Semantics for Hardware Prefetching

Georgios Vavouliotis[§] georgios.vavouliotis@huawei.com
Damjan Lampret[†] damjan.lampret1@huawei.com
Sara Friedman[‡] sara.friedman@huawei.com
Leeor Peled[‡] leeor.peled@huawei.com

[§]Computing Systems Lab, Huawei Technologies AG, Switzerland
[‡]Boole Lab, Huawei Tel-Aviv Research Center [†]Newton Research Center, Huawei R&D UK Limited

***Abstract*—Spatio-temporal prefetchers struggle with complex access patterns. Irregular prefetchers offer broader coverage but cannot capture access sequences that exhibit no address-space pattern; such sequences are derivable only from the address-generation code. Runahead prefetchers address part of this gap, but i) they cannot create slices for memory operations requiring impractical retired code flow history tracking, ii) they are inefficient at handling pointer-chasing patterns, and iii) they are limited by control flow complexity.**

This work proposes *SEMantic PREfetcher (SEMPRE)*, a hardware prefetch engine that addresses the challenges faced by prior prefetching schemes by exploiting the locality that arises when memory accesses recur due to similar program semantics. SEMPRE dynamically analyzes programs to learn their memory dependence chains and address calculation flows to construct storage-efficient code slices only for performance-critical memory operations, and uses them to prefetch complex access patterns. SEMPRE introduces a key innovation: it exploits long-range dependence prediction based on value equality to enable approximate slice creation for memory operations that would otherwise require impractical retired code history tracking.

Our evaluation, conducted using an industry-grade gem5 simulator, shows that SEMPRE outperforms (in geometric mean) previously proposed spatial, temporal, irregular, and runahead prefetchers by at least 5.5% and up to 9.6% on the SPEC INT 2017 benchmarks and by at least 2.1% and up to 14.8% on real-world mobile applications. We also demonstrate that SEMPRE reduces the address translation bottleneck, lowering L2-TLB MPKI by 47.9%–78.6% by prefetching across virtual page boundaries, thereby indirectly functioning as a TLB prefetcher.

***Index Terms*—prefetching, dependence prediction, irregular memory access patterns, code slices, memory level parallelism, mobile workloads, runahead execution, semantic locality, address translation, translation lookaside buffer, virtual memory**

I. Introduction

A significant part of the performance growth between CPU generations comes from advances in hardware prefetching [1]. Previously proposed prefetchers [2] analyze the memory access stream to identify patterns, ranging from sequential to traversals over linked data structures. Most prefetchers target *spatio-temporal locality*, relying on the premise that temporally and/or spatially adjacent accesses repeat [3]–[13]. Other prefetchers target irregular memory access patterns [14]–[19]. Runahead prefetchers speculatively execute address-generation code to prefetch accesses beyond the visibility of the out-of-order (OoO) window [20]–[25]; the address-generation code of a memory access is commonly referred to as a *code slice*, *i.e.*, the sequence of instructions responsible for generating the address of a memory access [20], [26], [27].

	PRE [23]	VR [24]	DVR [25]	SEMPRE
Handling of Massive History Requirements	✗	✗	✗	✓
Pointer-Chasing Patterns	✓*	✓*	✓*	✓
Complex Control Flows	✓	✗	✗	✓
Code Slice Optimization	✓	✗	✓*	✓
Independent of OoO Depth	✗	✗	✓	✓

TABLE I: Comparison with runahead schemes.

Although effective, state-of-the-art runahead prefetching schemes [23]–[25] face key challenges, summarized in Table I. Their three most critical limitations are: (1) they fail to prefetch for load operations requiring massive retired history tracking, *i.e.*, when the distance between two occurrences of the same memory operation is so large that it cannot be captured by history tracking structures with realistic sizes, (2) while they can capture pointer-chasing patterns, they need to recompute the entire dependence chain from scratch each time, which limits their effectiveness, and (3) they provide limited gains when the control flow becomes complex.

It remains unclear how these limitations manifest in emerging, real-world workloads and whether they fundamentally limit the applicability of runahead prefetching in modern systems. To answer this question, we analyze commercial mobile applications, which constitute a new and increasingly important workload class for CPU designs [28], while also considering the SPEC CPU 2017 benchmarks [29], which are commonly targeted by prefetching works.

Our analysis reveals three key findings. First, real-world mobile applications are dominated by long, control-heavy loops with poor vectorizability and extensive register spilling, which shifts dependences to memory and exacerbates the challenges faced by conventional runahead schemes. Second, the two workload classes differ sharply in slice diversity: a small number of unique code slices covers most memory accesses in SPEC, whereas mobile applications, whose code footprints reach over 2MB, exhibit an order of magnitude more, increasing the complexity of capturing recurring patterns in these applications. Third, the history required to reconstruct a slice scales the same way: SPEC workloads require modest code flow histories to construct accurate slices while mobile applications demand tracking substantially longer histories,

```
cb4f38   orr       x19, xzr, x28      ;Copy head ptr
cb4f3c   addxi_uop ureg0, x19, #8     ;Offset base ptr
cb4f40   ldp_uop   x1, x28, [ureg0]   ;Load data, next ptr
cb4f44   and       x20, x1, #0xFFF    ;Sanitize pointer
cb4f48   add       x21, x19, x20      ;Next node addr
cb4f4c   ldr       x22, [x21, #8]     ;Data from next node
```

Fig. 1: Pointer-chasing trace in a linked list traversal.

highlighting the difficulty of designing a slice-based prefetching scheme that remains both accurate and practical. Together, these findings explain why prior runahead prefetchers struggle on modern mobile workloads and call for rethinking their designs. The combination of long dependence chains, complex control-flow, and extended history tracking requirements directly stresses the key mechanisms these prefetchers rely on, fundamentally limiting their effectiveness.

This work demonstrates that exploiting *semantic locality* [26], where memory accesses recur under similar program semantics (*e.g.*, control-flow paths and data dependences), combined with long-range dependence prediction based on value equality, enables effective prefetching for complex access patterns that elude prior prefetching schemes. To that end, we design and propose *SEMantic PREfetcher (SEMPRE)*, a hardware-only prefetch engine that exploits code semantics to dynamically construct concise code slices and uses them to prefetch for complex access patterns. SEMPRE employs a flexible slice generation scheme that analyzes program code at runtime to identify dependence chains for address calculations and detect locality patterns through contextual similarities, offering advantages over Iterative Backward Dependence Analysis (IBDA) [30]. To keep slice generation and storage affordable in a realistic design, SEMPRE applies the mechanism selectively only to performance-critical memory operations, generating compact and optimized forecast slices which are stored and later used for prefetching.

To address the need for very large code history tracking in modern mobile applications, SEMPRE introduces a key innovation that *completes missing semantics using long-range dependence prediction based on value equality*. This concept exploits the result data of micro-operations (uops) to build code slices even when recorded history is not enough,[1] and SEMPRE realizes it through the *Out-of-History (OoH)* scheme, which dynamically builds approximate slices for memory operations that would otherwise require code flow history tracking beyond what is realistic to implement. More broadly, this new concept is not specific to prefetching: it enables approximate slice construction and applies to any slice-based microarchitectural technique.

We prototype SEMPRE in an industry-grade gem5 simulator [31] and show that it provides geometric mean (geomean) speedups over prior cache prefetchers and runahead schemes that range between 5.5% and 9.6% across SPEC INT 2017 [29] and between 2.1% and 14.8% across real-world mobile applications. SEMPRE also reduces TLB misses (47.9%–78.6% reduction) as it prefetches across virtual page boundaries and

[1]Micro-operation (uop) and operation are used interchangeably in the paper.

```
for (i=0; i<N; ++i) { // outer loop - regular, strided
  process_node(nodes[i])
}
```

```
void process_node(Node* n) {
 // thousands of instructions and many instructions via memory due
 // to register spill --> very common scenario in mobile applications
 // where every call is wrapped with register spill
 ....
 // complex, non-vectorizable pattern
 if(n->type == A){
  access_pattern_A(n);// irregular, data-dependent pattern
 }else if(n->type == B){
  access_pattern_B(n);// different irregular, data-dependent pattern
 }
 ... [more complex control flow] ... }
```

Fig. 2: Example from a mobile application.

issues speculative page walks that subsequently serve demand TLB accesses [32], [33]. Finally, we evaluate SEMPRE on an in-house industrial simulator and report consistent gains over a highly tuned baseline.

In summary, this paper makes the following contributions:

- We demonstrate that capturing the recurring memory access patterns of mobile applications requires (i) an order of magnitude more code slices than SPEC CPU benchmarks, and (ii) retired code flow histories of tens of thousands of uops, beyond what realistic hardware schemes can track.
- We introduce *long-range dependence prediction based on value equality*, a concept that uses the result data of uops to complete missing semantics and construct approximate code slices when the recorded history is insufficient. The concept can apply to any slice-based microarchitectural technique that needs approximate slice construction.
- We design SEMantic PREfetcher (SEMPRE), a hardware prefetch engine that exploits long-range dependence prediction to dynamically construct code slices for performance-critical memory operations and executes them to prefetch complex access patterns. Evaluated in an industry-grade gem5 simulator, SEMPRE outperforms prior cache prefetchers and runahead schemes across SPEC CPU 2017 and real-world mobile applications. We also show that SEMPRE reduces demand TLB misses by prefetching across virtual page boundaries.

II. Challenges in Runahead Prefetching

This section examines three state-of-the-art runahead schemes—Precise Runahead (PRE) [23], Vector Runahead (VR) [24], Decoupled Vector Runahead (DVR) [25]—and uses examples to explain their limitations, summarized in Table I.

Massive History Tracking. Previously proposed runahead schemes fall short for loads whose address-generating computation spans a very large dynamic distance. We define this distance as the number of uops on the minimal dependence chain that fully defines the address of a load, counting both register dependences and memory dependences reconnected through store-to-load pairs, *i.e.*, register spills. When this distance exceeds what realistically sized history structures can hold, the slice cannot be recovered: slice-based schemes such as PRE, VR, and DVR observe only a truncated prefix of the chain, while schemes that speculate forward on the instruction stream cannot run far enough ahead to expose

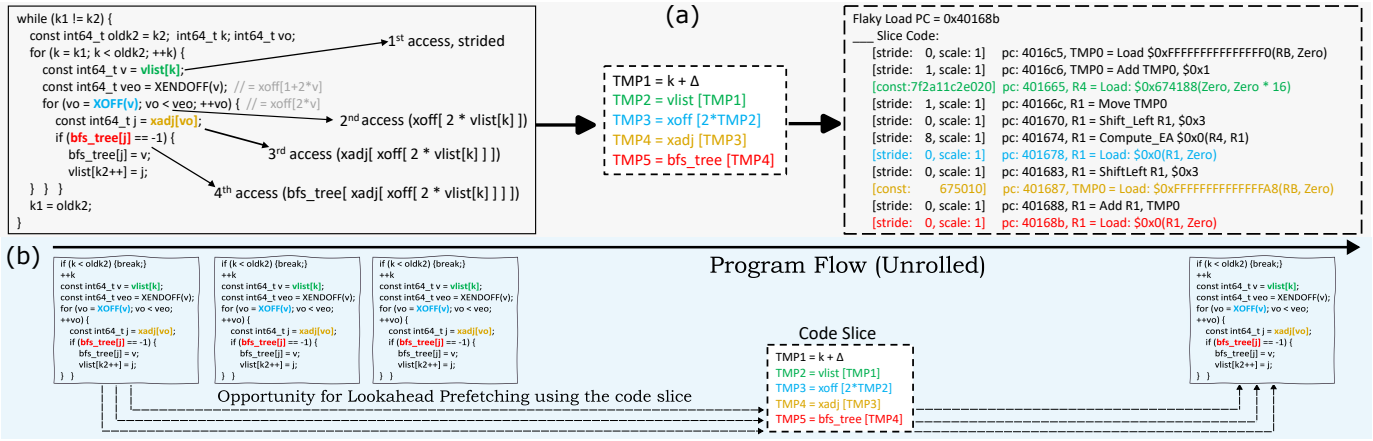

Fig. 3: Performance-critical bfs loop. (a) source code (left), slice pseudocode (middle), runtime-generated slice (right). (b) Dynamic flow unrolled to expose lookahead prefetching from the slice. Changing the stride Δ predicts accesses on far iterations.

the recurrence. Section III-C shows that such distances are very common in mobile applications. SEMPRE overcomes this limitation through long-range dependence prediction that exploits the result data of uops to construct approximate slices for memory operations that would otherwise demand massive history tracking. Section IV-C presents how SEMPRE realizes this concept in a microarchitectural scheme.

Pointer-Chasing Patterns. Figure 1 presents a pointer-chasing pattern typical of a linked list traversal where each step relies on the result of the previous, forming a chain of dependent memory accesses. While PRE, VR, and DVR can capture pointer-chasing patterns, advancing the chain requires re-executing its intermediate dependent loads: VR/DVR amortize this by vectorizing across iterations, but still execute the full dependent chain to reach a given depth. Our proposal, SEMPRE, addresses this limitation by memorizing intermediate pointer values across invocations without having to recompute everything from scratch. Section IV-F1 uses Figure 1 to explain how SEMPRE achieves this.

Complex Control Flows. When the control flow becomes complex, VR and DVR provide limited benefits as they rely on code being vectorizable. DVR supports branch reconvergence and nested vector runahead; we scope our claim to data-dependent divergence beyond those mechanisms. SEMPRE provides gains under complex and data-dependent control behavior by exploiting branch history information, as explained in Section IV-A, while continuously adapting its slices, preserving its effectiveness even as control and dependence flows evolve dynamically (Section IV-D, Section IV-E2).

Slice Optimization and OoO Depth. VR lacks slice optimizations (*e.g.*, slice pruning). PRE and VR are limited in range to what could fit in the enhanced OoO window, thus being constrained by finite OoO depth. SEMPRE optimizes slices to execute only relevant uops (Section IV-D) and is independent of the OoO depth.

A. Case Study from Mobile Applications

Figure 2 uses a code snippet from a real-world mobile application to illustrate the challenges such applications pose for runahead schemes, using DVR as a representative comparison; Section V presents all the applications considered. DVR fails on these patterns because (i) there is no local stride since the actual memory accesses inside `process_node()` are data-dependent and non-vectorizable, (ii) the irregular patterns do not appear in a way DVR could vectorize, and (iii) the DVR's Discovery Mode cannot link distant caller-level loads to deep, data-dependent memory operations in the callee. While DVR was evaluated on workloads with short, regular, and highly vectorizable loops [34], mobile applications feature long, control-heavy loops with extensive register spilling that shifts dependences to memory, breaking DVR's taint-propagation mechanism. Section IV-C1 shows how SEMPRE captures the patterns presented in the case study of Figure 2.

III. Correlating Program Semantics with Memory Access Patterns

This section highlights the potential of a prefetching scheme that exploits program semantics to overcome the challenges of runahead schemes, presented in Section II. Semantic locality [26] is a type of locality that arises when calculation steps of memory addresses repeat due to similar program semantics rather than mere spatial or temporal proximity. If semantic locality is properly extracted, it can enable effective prefetching even when spatio-temporal locality is absent. A scheme that correlates memory accesses through their dependence within the program's data layout and usage flow can enable prefetching for patterns that prior prefetchers cannot detect.

Semantic locality captures relations with strong recurrence since it describes program constructs. Extracting semantic locality requires following the set of uops that constitutes the relation between two memory addresses. We define a code slice as the minimal subset of dynamic code preceding a certain memory operation required to generate its memory address [20], [26], [27]. Code slices can be described through the data dependence chain that starts with the address calculation and goes backwards through all relevant sources at each step. *Extracting that form of locality can therefore be achieved by analysis of the address-generation code slice between two*

recurring instances of the same memory operation. Intuitively, having an accurate code slice for the performance-critical memory operations would enable effective prefetching even when there is no spatio-temporal locality.

A. Code Slices and Hardware Prefetching

Extracting complex patterns requires understanding of the program semantics. Figure 3 (a) shows this with a simple BFS code example [35]. The main *while loop* traverses an array of vertices, then an internal loop scans each vertex's outgoing edges to find its neighboring vertices and check their BFS depth. The top level access pattern is sequential, but the deeper levels are accessed through data-dependent patterns. The edge loop is also sequential but short, making the first edge of each vertex the critical element. These accesses have no spatial locality and very little temporal reuse. However, the middle box of Figure 3 (a) shows that the dependence chain within each iteration, whose accesses are increasingly critical to performance, can be represented using a short code slice.

Looking at Figure 3 (b), we observe how the extracted slice of Figure 3 (a)'s example can be exploited for prefetching. Due to the sequential nature of the top loop that exposes spatial locality within the first load in the slice, a change in the stride delta (Δ) can create a forecast slice that predicts accesses in the next iterations of the top loop. In other words, Figure 3 (b) shows how the slice can be used to issue useful prefetches for an arbitrary number of iterations in the future.

The example of Figure 3 shows how slices represent dependence chains in irregular structures and generate lookahead semantics for prefetching complex patterns. While this reveals the benefits of exploiting semantic locality for prefetching, our proposal is not restricted to such cases. By capturing spanning operations inherent to data structures, it generalizes to complex patterns beyond sequential code, as Section IV-F shows.

B. How Many Code Slices Are Needed?

Section III-A showed that code slices can capture the dependence chains behind complex patterns and generate timely prefetches for accesses that exhibit no spatio-temporal locality. Showing that slices are exploitable, however, does not imply that a realistic scheme can exploit them: a slice-based prefetcher should construct, store, and maintain a slice per targeted memory operation, so the number of unique slices a workload uses determines the required storage and, ultimately, the practicality of the design. Therefore, this section answers the following question: how many unique code slices are needed to cover the main data structures of modern mobile applications and standardized benchmarks? To do so, we use the slice generation model, presented in Section IV-B and Section IV-B1, without disclosing its details yet. Figure 4 shows the number of unique slices needed to cover all loads with a sufficient level of recurrence in memory-bound SPEC 2017 benchmarks [29] and mobile applications. Experiments are conducted using an industry-grade gem5 simulator [31]. Section V presents our baseline configuration and discloses the properties of the considered workloads.

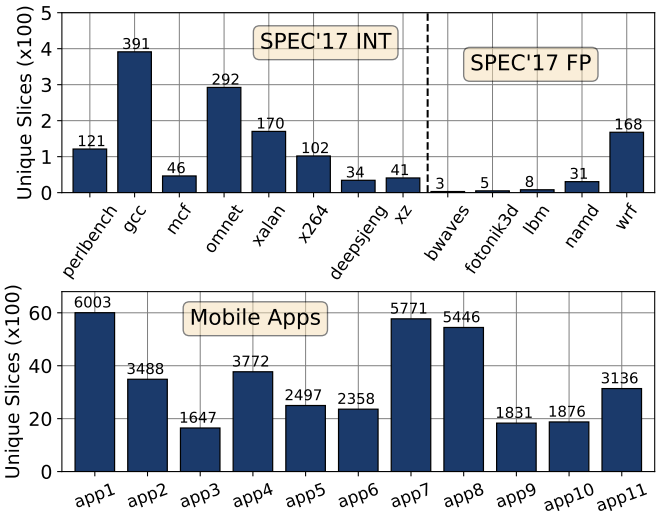

Fig. 4: Number of unique slices needed to cover all loads with sufficient recurrence (at least 6 dynamic occurrences of the load) in SPEC 2017 and mobile applications.

Figure 4 shows that the number of slices required to cover the recurring code flows of SPEC benchmarks is within a few hundred, indicating that a microarchitectural scheme could store and maintain those slices with reasonable storage budgets. Figure 4 also reveals that mobile applications have one order of magnitude more unique code slices than SPEC benchmarks, making it difficult for a realistic microarchitectural scheme to simultaneously store all these slices. However, any slice-based mechanism would build slices only for the most performance-critical operations. Finally, at any given program phase, only a subset of the slices will be actively used, thus reducing the number of slices that need to be stored and maintained by a scheme (*e.g.*, prefetcher) at any given time.

C. How Much Code Flow History Is Needed?

This section couples the results of Section III-B by measuring the average code flow history length that needs to be tracked between two consecutive instances of the same load operation. This refers to the number of uops on the minimal dependence chain that fully defines the address of a load, where the chain includes both register dependences and memory dependences reconnected through store-to-load pairs, as defined in Section II. In other words, we count the minimum number of uops that need to be tracked to build an accurate slice for any given load. Figure 5 shows results for both SPEC benchmarks and mobile applications.

The load operations of some SPEC benchmarks (*e.g.*, `mcf`) require very short history lengths to build complete slices, while others (*e.g.*, `omnet`) need thousands of uops to be stored to generate accurate slices. Looking at Figure 5, we observe that the mobile applications require massive code history tracking to enable accurate code slice creation. Notably, the application with the lowest average code history length (`app7`) requires more history than the SPEC benchmark with the highest demand for history tracking (`omnet`). The main takeaway

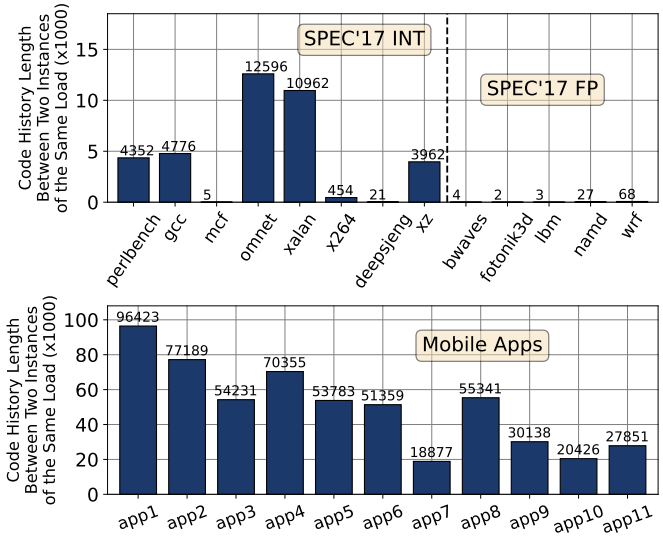

Fig. 5: Average number of uops between two instances of the same load in SPEC 2017 and mobile applications.

of this study is that real-world mobile applications and some SPEC 2017 benchmarks require excessive retired code flow history tracking to build accurate slices, posing significant storage challenges for schemes that want to dynamically build, store, and exploit code slices for microarchitectural optimizations (*e.g.*, prefetching).

IV. Exploiting Program Semantics for Hardware Prefetching

This section presents *SEMantic PREfetcher (SEMPRE)*, an engine that exploits program semantics for prefetching. A key novelty of SEMPRE is using value-based dependence prediction to create approximate slices for memory operations that would otherwise require impractical code flow history tracking. Figure 6 shows the architecture of SEMPRE, its components, and its high-level operation. SEMPRE operates in three stages. First, flaky load detection selects which loads to prefetch for, based on load context (PC and branch history), load hotness, and cache behavior (Section IV-A). Second, for each flaky load, SEMPRE constructs and validates a code slice from the retired uop stream (Section IV-B–Section IV-D). Third, stable slices are executed by an engine of SEMPRE to issue prefetch requests (Section IV-E). Overall, SEMPRE consists of seven components:

- ***Flaky Load Detector*** identifies frequently executed loads with high cache miss rates (Section IV-A).
- ***History Queue*** stores the retired code flow for accurate slice creation. Each uop's metadata is written into the History Queue as the uop commits from the ROB head: each entry records the uop (its opcode and register sources/destination) and the uop's result data (used in Section IV-C for approximate slice creation). The History Queue is only used for slice creation and does not store slices.
- ***Unique Dependence Queue (UDQ)*** is a victim buffer for evicted History Queue entries with single register dependence. SEMPRE uses UDQ to build approximate slices for loads that would otherwise need unrealistic History Queue sizes to build slices for (Section IV-C).
- ***Walker*** is an FSM that is responsible for generating and validating code slices (Section IV-B) using the History Queue and the UDQ. The Walker ensures that slices are concise and that executing them will generate useful prefetches.
- ***Slice Array*** stores validated (stable) slices coupled with metadata used for adapting and optimizing prefetching across execution phases, where the usefulness of a slice at generating accurate prefetches may change.
- ***Slice Execution Engine (SEE)*** executes validated code slices to generate and issue prefetch requests (Section IV-E).
- ***Update Queue*** tracks issued prefetches to train SEMPRE and adapt its strategy across phases (Section IV-E2).

A. Flaky Load Detection

The *Flaky Load Detector* is responsible for identifying the loads that are critical for performance, *i.e.*, the loads for which SEMPRE will construct a slice that will be later used to issue prefetches. To do so, *Flaky Load Detector* uses a table named Flaky Load Table (Ⓐ in Figure 6) that tracks loads by storing a tag and two metadata counters, using the pLRU replacement policy. The tag is a combination of the program counter (PC) and a branch history register (BHR) that tracks up to 6 recent branches. Each branch is represented by the lowest 4 bits of its PC, with the least significant bit XOR-ed with the binary outcome of that branch. Together, the PC and the BHR represent an instance of a load within a specific program context. Following this method, SEMPRE distinguishes between different occurrences within nested loops or complex control flows, which may affect the access pattern and the generated slice. Note that the PC and BHR are concatenated and hashed to create an index that identifies the load across all modules of the prefetcher. The BHR contributes 24 bits (six branches, 4 bits each) and is combined with the load PC bits, so PC aliasing is low at the table sizes used (Table V). In this way, SEMPRE captures patterns originating from different control flows.

Regarding the metadata counters, each Flaky Load Table entry stores the (1) *hotcnt* that counts the number of load occurrences and the (2) *misscnt* that counts how many times this load missed in L1D. A load is considered flaky if the corresponding *hotcnt* and *misscnt* counters exceed pre-defined thresholds. For each flaky load, SEMPRE constructs a slice that is eventually stored in the Slice Array (Ⓑ in Figure 6).

B. Slice Generation

This stage determines how each flaky load's address is produced by analyzing the retired uop stream. Figure 3 assumes that the slice is given to show the potential of exploiting code semantics for prefetching. However, tracking all dependence chains for a given load would construct a graph of uops that may span back to the beginning of the program. This section presents an effective slice generation model and its properties. History tracking is limited by breaking the dependence chain in the following cases: 1) constant values remaining static

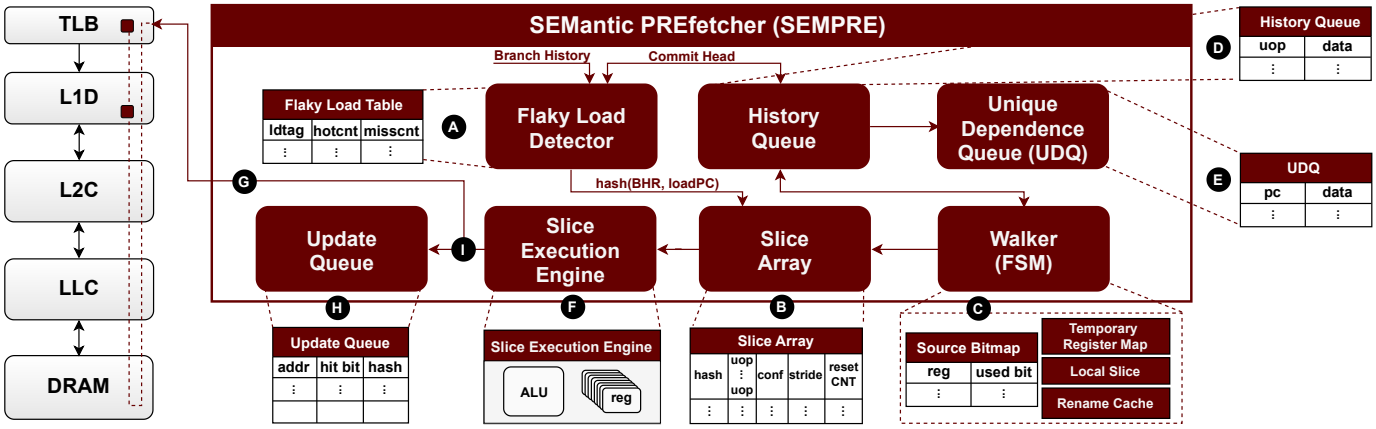

Fig. 6: Design overview of *SEMantic PREfetcher (SEMPRE)* with operation steps. SEMPRE uses the Commit Head (ROB head) to access the retired code stream. The data fields of the History Queue and UDQ correspond to the result data of uops.

during analysis, 2) strided values proven to have a constant stride or that are produced by a simple add/sub operation with constant or immediate sources, and 3) when analysis encounters the same operation it started from, the dependence chain can stop as it would repeat. Linked data structures may iterate a few times to create a deeper chain.

Before a slice can be used to generate prefetches, it needs to be sanitized. First, the destination registers are replaced with temporary ones (guaranteed not to be used by the original code) and their occurrences as sources within the slice are renamed accordingly. Second, memory writes are removed from the slice. Since the code was generated through a dependence chain, all writes to memory were added to resolve younger loads reading from the same address. Hence, a simple memory renaming can be used to replace such store-load operations with move operations to a reserved register.

After sanitization, the slice can be used to generate prefetches. To do so, any detected stride is extended by a lookahead factor, *i.e.*, if an operation in the slice was detected to induce a stride of N, that stride is replaced by $L \times N$ where L is the lookahead applied for that slice. This lookahead is initialized to point a few iterations ahead, but its value can be dynamically adapted (Section IV-E2) to allow further lookahead based on the average hit depth of a given slice.

1) Realization of the Slice Generation Model: This section explains how SEMPRE realizes the previously described slice generation model. SEMPRE constructs a slice for each load identified as flaky (Section IV-A). Each slice consists of a selected code path subset prior to the corresponding flaky load. To generate the slice, it uses a Walker FSM (C in Figure 6) that walks the History Queue (D in Figure 6), which is a queue that tracks the program code flow at retirement.

The Walker traverses the History Queue from the youngest uop (the flaky load) to the oldest and constructs the code slice. To track data dependences, the Walker (C in Figure 6) uses (1) a *Source Bitmap* which assigns one bit per logical register to track the active sources (only general-purpose and flags registers), (2) a *Rename Cache* that tracks memory operations for potential memory renaming [36], *i.e.*, tracks which stores and loads have the same memory address, and (3) a *Temporary Register Map* which is a renaming scheme that tracks memory locations replaced with temporary logical registers; these are special registers and not registers from the general purpose register file. The Walker can store up to 16 uops that serve as a local copy of the slice (Local Slice) during construction, has an index pointing to the History Queue for the traversal, and a counter of temporary registers used for memory renaming.

The Walker first sets the bits representing the load sources and then traverses the History Queue backwards (from youngest to oldest uop). On each uop that writes to a register in the bitmap, the Walker performs the following steps: 1) pushes the uop to its local slice, 2) clears the destination register from the sources bitmap marking that its producer has been added, 3) sets all registers corresponding with the current uop sources to ensure that older uops producing these sources will be added, 4) records the result data of the uop; this will be checked for constants or strides during the next phases.

Loads that are added to the slice record their address and local slice index in the Rename Cache, a stack buffer structure that can host 16 addresses. The Walker then performs memory renaming whenever an older store is observed (further along the walk) that matches an address in the Rename Cache. Renaming is done by extracting the index of the matching load from the structure and replacing both the store and the load operations in the Local Slice with a move to and from (respectively) an available temporary register. Note that reducing the store/load pair further by moving the store data directly to the load destination is not possible, since the load destination register may be reused between the store and the load (and therefore override the data).

The Walker completes the traversal upon (1) reaching the tail of the History Queue (no encounter of the same load), (2) encountering the same load (complete slice), or (3) when there are no longer valid sources marked in the Source Bitmap. Upon completing the traversal, the slice has been generated but not yet validated. Section IV-D details the code slice's lifetime.

C. Out-of-History (OoH) Slices

Section IV-B assumes that during a walk for a flaky load, the Walker would find in the History Queue a previous instance of the same load. This mostly happens for the SPEC 2017 benchmarks that require short history tracking lengths, as shown in Figure 5. In such cases, there is all information required in the History Queue to build complete code slices by following the slice generation algorithm of SEMPRE, described in Section IV-B.

Modern mobile applications, like the ones used in Section III-C, require massive History Queue sizes to build complete code slices, as shown in Figure 5. However, the size of the History Queue is limited in realistic hardware implementations. SEMPRE addresses the need of applications for larger History Queue sizes by applying a new concept that completes missing semantics using long-range dependence prediction based on value equality, *i.e.*, it exploits the result data of uops without increasing the size of the History Queue. In practice, SEMPRE uses the actual result data of uops to create approximate code slices for flaky loads with recurrence that realistic History Queues cannot cover. We refer to this category of flaky loads as *Out-of-History (OoH)* loads. SEMPRE exploits this new concept for enhancing the effectiveness of slice-based prefetching. However, this concept is not limited to prefetching, as it can be applied to any slice-based mechanism that needs approximate slice construction.

A flaky load is classified as OoH only if the Walker reaches the tail of the History Queue (no encounter of the same load), meaning that there is not enough retired code history in the History Queue to construct a complete code slice. One option is not to build code slices for OoH flaky loads. Section V-D shows that not building slices for OoH loads significantly harms the performance of SEMPRE, highlighting the importance of the OoH scheme.

The OoH scheme uses the Unique Dependence Queue (UDQ) that stores evicted History Queue entries with single register dependence. The OoH scheme operates as follows. First, the Walker traverses backward the History Queue and looks for uops with single register dependence. For uops with single register dependence, it uses the result data of the uop, which is stored together with the uop in the History Queue (Ⓓ in Figure 6) to search UDQ for possible hits (Ⓔ in Figure 6). Upon UDQ hits, SEMPRE creates a slice that starts from the uop whose data hits in UDQ until the flaky load itself. UDQ is practically used to approximate the start of the slice. The actual slice is eventually built using the History Queue. The Walker considers only single-register-dependence uops as candidate anchors and stops at the first one encountered during the backward walk whose result value hits in the UDQ. A single hit is therefore sufficient to anchor the slice, and the number of single-register-dependence uops inspected before a hit is example-specific rather than fixed. Figure 7 depicts this operation with an example, assuming that there is only one instance of flaky `ldr x10, [x9, #1232]` in the History Queue; thus, this load is OoH. As the Walker traverses

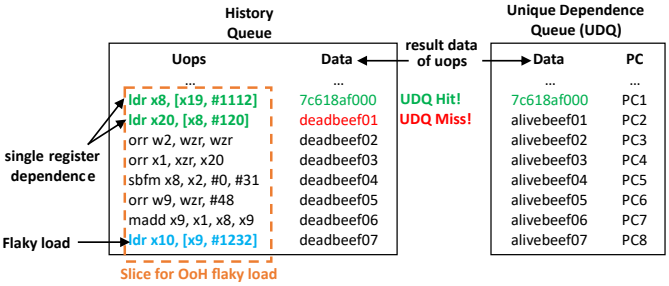

Fig. 7: Example of slice construction for an OoH flaky load. Walking backward from the load, the Walker inspects uops with a single register dependence and anchors the slice at the first one whose result value hits in the UDQ. Here the younger candidate `ldr x20` misses, so the walk continues to `ldr x8`, which hits and becomes the anchor.

backwards the History Queue for uops with single register dependence, it first encounters `ldr x20, [x8, #120]`, but this load does not hit in UDQ, thus failing to trigger a slice creation. Next, the Walker encounters `ldr x8, [x19, #1112]`, which has a single-register dependence and hits in UDQ (using the result data of the uop for lookup). In this case, the Walker creates a slice that starts from the uop `ldr x8, [x19, #1112]` until the flaky load `ldr x10, [x9, #1232]` using the History Queue. If there are no uops in the History Queue with single register dependence, or there are but none of them hit in UDQ, SEMPRE does not build a slice for the corresponding load. Note that SEMPRE looks for only one match in UDQ and ignores slices with multiple dependences per slice because we found that handling multiple dependences per slice yields gains too small to justify the added complexity over the single-match case.

1) Case Study from Mobile Applications: Figure 2 and Section II-A use a snippet from a mobile application to highlight a key inefficiency of prior runahead schemes. SEMPRE succeeds in capturing the patterns of Figure 2 because: i) the OoH scheme uses the data value loaded from `nodes[i]`—which occurs a few thousand instructions before the complex inner accesses—as a trigger for the downstream slice. ii) This natural separation of a few thousand instructions provides an effective prefetch distance; by the time `nodes[i+L]` is loaded, its dependent memory operations are prefetched just in time. iii) the OoH scheme builds an approximate slice that spans across the function call, reconstructing the dependence chain via value matching—the UDQ value-equality check described above, in which the recurring result data of a single-register-dependence uop (here, the data value loaded from nodes[i]) hits in the UDQ and anchors the slice—even when intermediate instruction history is unavailable. iv) When `nodes[i+L]` is accessed in future iterations, SEMPRE triggers prefetches for its complex dependent patterns, ensuring that the prefetched data arrives when needed. This allows the OoH scheme to uncover and prefetch across call boundaries and long dependence gaps that prior runahead schemes cannot handle (Section II).

D. Slice Lifetime

Sections IV-B and IV-C describe how SEMPRE generates slices but not when they are used for prefetching. This section explains the lifetime of a code slice, from its creation to the point where it can be used to trigger prefetch requests.

Figure 8 illustrates the life cycle of a slice in steps. When the Flaky Load Detector characterizes a load as flaky (❶ in Figure 8) by following the operation described in Section IV-A, an entry in the Slice Array is allocated and its state is set to `Gen` ❷. At this point, a Walker is assigned to construct the slice of this load (Section IV-B, Section IV-C). Upon completion, SEMPRE switches the slice to `Validate` state ❸. During this phase, when the same load context is encountered again, SEMPRE assigns a Walker to perform the walk once more to validate that the slice did not change. The slice remains in the validation phase for several encounters (empirically selected to perform 3 validation rounds) to ensure the slice is stable and to identify constant/strided values. The slice generation process can be aborted in any of the following cases: (1) slice is inconsistent during validation due to insufficient context length or no recurrence, (2) slice is too long for storing (length 16 is empirically selected), (3) need for too many temporary registers (limit is empirically set to 8), or (4) we are out of history since there is a very long dependence chain that exceeds the size of the History Queue and the OoH scheme (Section IV-C) cannot create a slice. Slices that do not pass the validation step start the entire process from scratch ❹. Note that resets during slice construction are considered transient, meaning that a later attempt may construct a useful slice.

The slices that pass all validation rounds are considered stable ❺ and pass to `Trim` phase; this is the only step allowed to modify the slice ❻. During the `Trim` phase, the walk is performed again but stops tracking sources when reaching constants or strides that were discovered during the validation passes and replaces them with a simple immediate move or add/sub. As a result, some parts of the data dependence flow may be removed from the slice, resulting in a smaller slice that reduces the storage footprint of Slice Array (Section V-E4).

`Trim` phase also renames the destinations to temporary registers to avoid side effects. The Walker traverses the constructed slice and converts each destination register to the next available temporary register. The conversions are recorded in the Temporary Register Map (Section IV-B). During the following traversal steps, all younger slice instructions will rename any matching sources to read from the corresponding temporary register. After trimming, the code slice reaches `Ready` state ❼ and can be used for prefetching.

Figure 9 presents an example of slice generation and life cycle over code striding across an array that requires double dereference. The dependence chain is discovered by walking the History Queue, as shown on the right side. The unrelated multiply operation is removed, but all other operations are identified as part of the dependence chain. The intermediate slice, *i.e.*, the raw slice without optimization, remains consistent during several validation phase iterations. During this

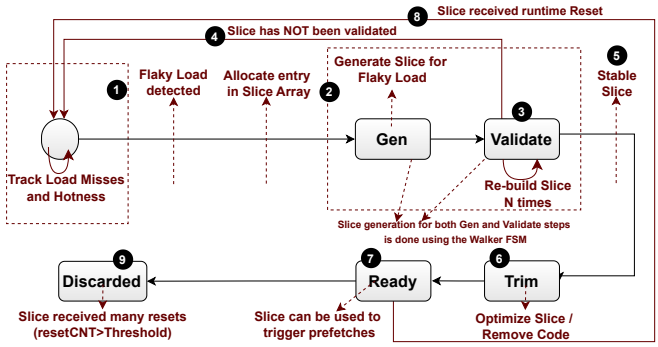

Fig. 8: Life cycle of a code slice.

period, the add operation is detected as a stride of one and the two middle loads are identified as constants. Since R2 is constant, the `Trim` phase replaces it with a move and stops processing its dependences (thereby also eliminating the R4 load). The final slice is therefore three operations long. The operation is the same for OoH slices (Section IV-C).

E. Prefetching with the Slice Execution Engine

For `Ready` slices (❼ in Figure 8) that can be used for prefetching, each encounter with its load context (PC, BHR in Section IV-A) activates the SEE (Ⓕ in Figure 6) to execute the corresponding slice and produce prefetches. The SEE is a sequencer with a 2-stage pipeline. It does not have a front-end and does not support control-flow instructions since SEMPRE uses BHR information to capture patterns originating from different control flow. The SEE i) consists of an ALU for arithmetic operations and 8 registers, ii) does not have caches, iii) is asynchronous to the main pipeline, and iv) does not share resources with the main pipeline because it harms performance due to added stress over critical resources.

The machine width is a large factor for compute-bound applications. For memory-bound applications, the width of the machine is not a major factor. In our experiments, the IPC for memory-bound applications is in the range of IPC 0.44–1.50. To keep up with the modern high-throughput processors (we tested 6- and 8-wide), we found that an IPC of 0.5 is sufficient for the SEE to produce timely prefetches for 97% of the simpoints. To keep SEE simple and ease timing closure, we run it at half the CPU clock frequency. This allows for a straightforward 2-stage pipeline design. A single slice can have only one memory access in flight, so it spends about 90% of its cycles waiting for memory responses. The SEE therefore round-robins across up to 10 active slices, which keeps it issuing one uop per cycle.

During the slice execution process, sources marked as constants use the recorded constant value, but operations marked as having a stride are adjusted by having their stride value multiplied by a dynamic lookahead factor. This lookahead is initialized for each code slice to one, meaning that SEMPRE executes the slice as observed, without any extrapolation (thereby performing the address computation of the next iteration). However, for slices that are proven to provide non-timely prefetches (Section IV-E2), the lookahead factor will

increase gradually up to a maximum number. This maximum number is empirically set to 64 to allow prefetching far enough ahead of time, but not too far as to exceed the cache lifetime. Note that all strides within a slice are always multiplied by the same lookahead factor so that the ratios between strides are always kept as they were detected over a single iteration.

After following this process, the final operation in the slice is a copy of the original load that the slice was constructed from, but since any strided sources were enhanced to apply a lookahead over their strides, the load address would belong to some future iteration. This is actually the prefetch generated by executing this slice and is sent to the memory hierarchy as a prefetch request (step G in Figure 6). The prefetch is also inserted in the Update Queue along with its predicting slice hash for usefulness tracking (step ❶ in Figure 6). Note that SEMPRE prefetch operations have lower priority than demand loads in shared resources (*e.g.*, ports, MSHRs) and can be stored anywhere in the cache since SEMPRE does not use a dedicated part of the cache for storing its prefetches.

1) TLB Prefetching: SEMPRE's prefetches that cross pages might not find the translation of the page where the prefetched block resides in the TLB. SEMPRE triggers speculative page walks for prefetches that miss in the TLB, acting as a TLB prefetcher. Section V-A1 shows that SEMPRE reduces the address translation bottleneck by saving demand TLB misses.

2) Adaptive Training and Throttling: To ensure that SEE issues useful prefetches, SEMPRE uses the Update Queue, a FIFO structure that tracks the issued prefetches together with metadata for training and adapting decisions to different phases [37]. Update Queue entries (H in Figure 6) store (i) the prefetch address, (ii) a *hit bit* annotating whether the prefetched line served at least one demand access, and (iii) a *hash* to index the Slice Array entry that corresponds to the slice that was executed to generate this prefetch.

Regarding the operation, demand L1D accesses look up the Update Queue for possible hits. Upon an Update Queue hit, the `hit bit` of the hit entry (H in Figure 6) is set to 1, indicating that this prefetch saved at least one demand access during its lifetime in the Update Queue. The `hash` stored in the hit entry is used to index the Slice Array and update the confidence counter (`conf` in Figure 6 B). The reward function for the hit entry operates as follows: 1) demand L1D accesses that hit in one of the first D entries (D is set to 3) in the Update Queue indicate that the corresponding prefetch was late, thus receiving a negative confidence update, and 2) the rest of the Update Queue hits trigger a positive slice confidence update in the Slice Array. Finally, prefetches that get evicted from the Update Queue without providing any hit during their lifetime, *i.e.*, `hit bit` is zero, are considered useless and trigger a negative confidence update for the corresponding slice using the hash of the evicted Update Queue entry.

Adaptive Lookahead. SEMPRE dynamically adjusts the lookahead factor to adapt to different workload behaviors. It monitors the hit depth, *i.e.*, how far a prefetched block has advanced through the Update Queue before a demand consumes it, and adapts the prefetch distance accordingly.

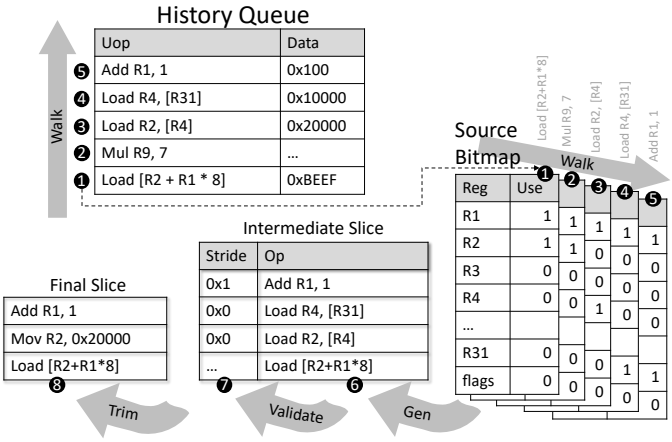

Fig. 9: History Queue is walked backward from the flaky load. After the walk, a verbose slice is generated (`Gen` state) including only the dependence chain (R9's multiply is dropped). Stride/Const values are populated during `Validate` phase. The final slice is formed after the `Trim` phase, in which R4 was found to be constant.

A demand that hits near the front of the Update Queue indicates that the prefetch arrived late, so SEMPRE increases the lookahead factor for the corresponding slice; prefetches evicted from the Update Queue without a hit indicate that SEMPRE ran too far ahead, so it decreases the factor. This adaptive tuning allows SEMPRE to maintain an effective lookahead distance across different phases.

Runtime Resets. When the confidence of a slice (`conf` in Figure 6 B) falls below a threshold after repeated negative updates, it is deemed useless for the current phase and a runtime reset is triggered for it (❽ in Figure 8), meaning that this slice needs to pass the `Gen`, `Validate`, and `Trim` phases again to be used for generating prefetches. This process allows regenerating the slice in case the code flow changes compared to when it was originally constructed, while ensuring that slices that are proven to issue useless prefetches are not used until they prove useful again (if they ever do).

Permanently Discarding Slices. SEMPRE counts runtime resets for each slice in the Slice Array (`resetCNT` in Figure 6 B). Slices with more than N (empirically set to 25) resets are considered unstable. These slices rarely issue useful prefetches while wasting a lot of resources, thus their state becomes `Discarded`, preventing their reconstruction (❾ in Figure 8); this state is held in the Slice Array entry and is not persisted, so if the entry is evicted, a subsequent flaky detection of the same load rebuilds and re-validates the slice from scratch.

F. Patterns and Program Structures

Any given data structure has a set of operations that define all forms of traversals across it. The slices of SEMPRE accurately represent the dependence chain of irregular data structures; these slices generate lookahead semantics that can be used for driving prefetching for complex patterns.

1) Case Study: Figure 1 presents a pointer-chasing pattern typical of a linked list traversal. The code loads a pointer

OoO Core	512-entry ROB, 8-wide fetch, 6-wide retire, TAGE and iTAGE, 2.86GHz
ITLB/DTLB	64-entry, 1-cycle latency, pLRU
L2-TLB	1280-entry, 8-cycle latency, pLRU
L1I	128KB, 8-way, 3-cycle latency, pLRU
L1D	64KB, 4-way, 3-cycle latency, pLRU
L2C	1MB, 8-way, 11-cycle latency, BRRIP
LLC	3MB, 12-way, 15-cycle latency, BRRIP
Memory	8GB LPDDR5, 4 channels, 5500 MTPS

TABLE II: System Configuration.

from memory (`ldp_uop x1,x28,[ureg0]`), applies an offset to access the pointer to the next node, computes the address of the next node (`add x21,x19,x20`), and performs a dependent memory access (`ldr x22,[x21,#8]`) to load data from the next node; this load is flaky. Each step relies on the result of the previous, forming a chain of dependent memory accesses. SEMPRE effectively handles pointer-chasing patterns like that. First, it identifies the traversal portion of a slice to walk through the linked list (same applies to other pointer-based data structures). By isolating the traversal, SEMPRE can treat it separately from the rest of the slice and enable targeted prefetching. After traversal isolation, SEMPRE proactively executes the traversal for `N` steps to walk ahead in the chain and compute a future memory address. The result is stored in the slice context. At the next slice execution, instead of repeating all `N` traversal steps, SEMPRE resumes from the saved address and advances with a single additional dereference. This greatly reduces redundant work and advances over prior runahead schemes that must execute all intermediate iterations to peek ahead, shown in Section II.

To see why this keeps SEMPRE ahead even on a wide core with little work between iterations, consider a pure pointer-chase where each node occupies one cache line. With no prefetching, the core cannot exploit its issue width since there is no independent work to overlap, so it walks the chain at roughly one DRAM access per node. When the core stalls on the first node, SEMPRE isolates the traversal and runs it N nodes ahead, with each access missing to DRAM but installing the node in the cache. By the time the core reaches those nodes, they hit in the L2C rather than missing to DRAM, so the core's per-node latency drops. Because the now-accelerated core would otherwise catch up, SEMPRE's lookahead factor (Section IV-E2) is sized so that SEMPRE's DRAM-latency walk stays far ahead to keep fetching nodes before the core demands them. The cost to SEMPRE is a single dereference per node; the benefit to the core is converting each per-node DRAM miss into a cache hit. This is how SEMPRE sustains an N-node lead at one load per iteration while the core, absent SEMPRE, would otherwise traverse at full latency.

The true focus of SEMPRE is to prefetch for the leaf flaky load (`ldr x22, [x21, #8]`) since it is a performance-critical load. However, this load depends on the traversal result, explained above, and SEMPRE cannot access it without resolving the pointer chain. Thus, SEMPRE builds a slice for this flaky load using the traversal merely as the necessary path to reach it and enable effective prefetching.

	Code Footprint (MB)	Data Footprint (MB)	Description
app1	2.2	7.1	Client for a search engine
app2	1.0	5.9	Social media-online video app
app3	2.0	4.3	Chat app
app4	2.3	5.5	Online shopping app
app5	1.5	15.0	News app
app6	1.1	4.9	Content creation app
app7	1.2	4.4	Mobile game (rpg)
app8	0.5	2.0	Mobile game (role play)
app9	1.2	2.2	Mobile game (strategy)
app10	1.0	4.7	Mobile game (racing)
app11	1.4	8.4	Mobile game (moba)

TABLE III: Properties of the considered mobile applications.

V. Evaluation

Our evaluation uses an industry-grade gem5 [31] with ARM ISA and a non-inclusive, non-exclusive (NINE) cache hierarchy. Table II summarizes the modeled system. Section V-E5 also evaluates SEMPRE on an in-house industrial simulator.

Workloads. We use all SPEC CPU 2017 [29] and a set of real-world applications drawn from market research on commercial mobile products. We refer to these applications as mobile applications and keep their names anonymous for business reasons. The data and code footprints and a description of the mobile applications are reported in Table III. For each application, a performance-modeling team identified representative usage scenarios targeting user experience, along with the corresponding code regions and inputs; each scenario reflects common user behavior. The applications are captured as user-level execution, so they only run user code. We select regions of interest with SimPoint [38], using a warmup window for all on-chip resources before a measured window; measured regions consist of at least 40M instructions, with most SimPoints exceeding 100M instructions, and results are weighted across SimPoints per application. We apply the identical SimPoint-selection, warmup, and weighting methodology to the SPEC and mobile workloads, so the two suites are characterized on equal terms. Consequently, the long recurrence distances in Figure 5 reflect application structure, not system activity or elided code: message/event loops, view hierarchies, and interpreted or JIT-compiled runtimes traversing deep object graphs let a logical load recur only after tens of thousands of intervening instructions [39].

A. Comparison with Hardware Prefetchers

This section compares SEMPRE with previously proposed cache prefetchers over a baseline without prefetchers, presented in Table II. Specifically, we evaluate four prior spatial, temporal, and irregular prefetchers: (1) Indirect Memory Prefetcher (IMP) [19], (2) Signature Path Prefetcher (SPP) [40], (3) Access Map Pattern Matching (AMPM) [41], and (4) Correlated Miss Caching (CMC) [42], [43], a temporal prefetcher equipped with 512KB of storage to extract its full potential. All competing prefetchers operate with virtual addresses. Therefore, prefetches need to go through the TLB hierarchy and potentially trigger speculative page walks.

Figure 10 presents the performance of SEMPRE, IMP, SPP, AMPM, and CMC across the SPEC benchmarks and

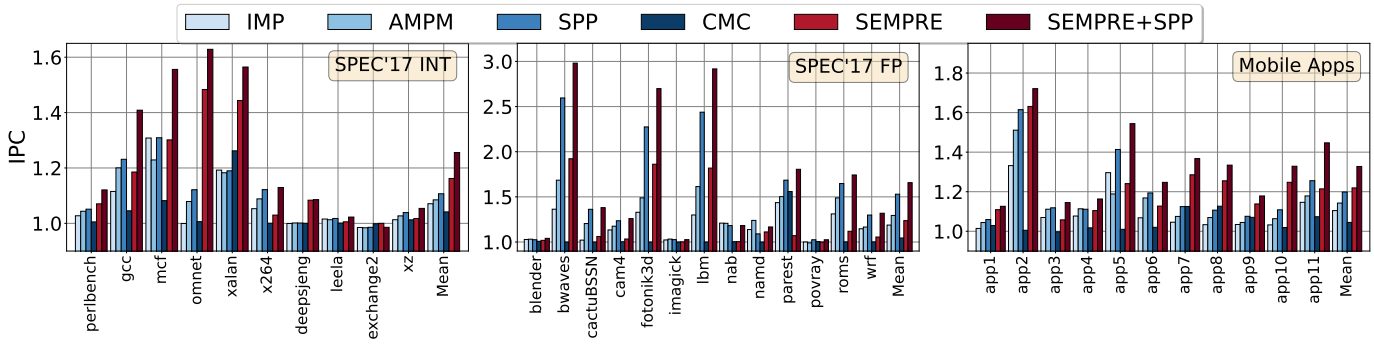

Fig. 10: Performance of SEMPRE and previously proposed prefetchers across SPEC 2017 (INT, FP) and the mobile applications.

the mobile applications. SEMPRE yields higher gains than all competing prefetchers for most SPEC INT. Specifically, SEMPRE outperforms IMP, SPP, and AMPM by 9.1%, 5.5%, and 7.7%, respectively. The highest gains are offered to `omnet` and `xalan` because capturing the irregular patterns of these benchmarks requires tracking a small number of code slices, as shown in Figure 4, and most of these slices are stable and can be used for prefetching, as Section V-E1 shows.

The mobile applications show similar trends with SPEC INT. SEMPRE outperforms IMP, SPP, and AMPM by 11.4%, 2.1%, and 7.6%, respectively, because it correlates program semantics with complex patterns that other prefetchers, lacking semantic analysis, fail to detect.

On SPEC FP, spatial prefetchers like SPP and AMPM outperform SEMPRE on many benchmarks. These benchmarks are dominated by spatially correlated accesses that a spatial prefetcher detects, while exposing few of the semantically recurring patterns SEMPRE targets. The same effect explains the handful of SPEC INT and mobile benchmarks where SPP outperforms SEMPRE (*e.g.*, `gcc`, `app5`, `app6`). This is by design: SEMPRE ignores spatial patterns rather than spending slice-generation and storage resources on accesses a spatial prefetcher already covers, and instead targets complex patterns that lack spatio-temporal locality. The two are therefore complementary rather than competing, as we show next.

To support our claim, we combine SEMPRE with SPP in Figure 10. SEMPRE+SPP achieves higher speedups than either SPP or SEMPRE alone across all workloads. Representative examples are `mcf`, `lbm`, `app2`, and `app5`, among others. This improvement arises because SPP targets spatial patterns, while SEMPRE targets complex patterns that prior prefetchers cannot detect. We observe similar behavior when combining SEMPRE with other spatial/temporal prefetchers.

1) Impact on TLB Hierarchy: Since all evaluated prefetchers issue requests in the virtual address space, a prefetch whose target page has no TLB entry triggers a speculative page walk [32]. The corresponding translation is stored in the TLB and potentially serves later demand accesses, so every prefetcher performs TLB prefetching [44], [45]. Figure 11 reports DTLB misses per kilo instructions (MPKI) and L2-TLB MPKI for IMP, SPP, and SEMPRE; we omit AMPM and CMC from this study due to low performance gains.

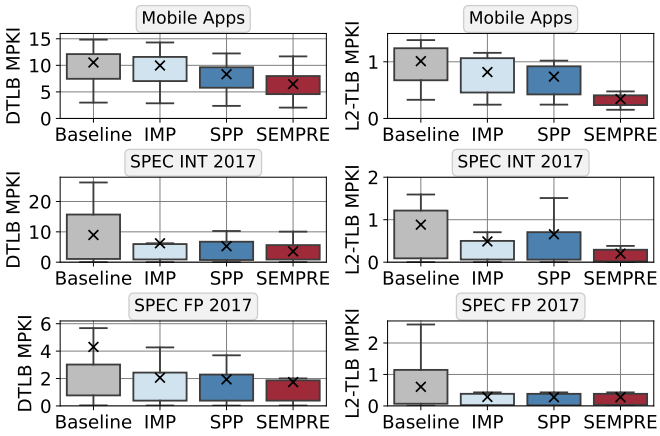

Fig. 11: Impact of SEMPRE and prior cache prefetchers on DTLB MPKI and L2-TLB MPKI. Lower is better.

SEMPRE is effective at TLB prefetching since it reduces average DTLB MPKI for SPEC INT, SPEC FP, and mobile applications by 53.1%, 58.8%, and 38.9%, respectively. The reduction in L2-TLB MPKI is 78.6%, 47.9%, and 63.5% for SPEC INT, SPEC FP, and mobile applications, respectively, meaning that the system experiences fewer page walks with SEMPRE. DTLB MPKI is more sensitive to page-cross prefetching than L2-TLB MPKI because the DTLB is smaller and translations fetched by page-cross prefetches are stored in both the DTLB and L2-TLB; thus, a useful page-cross prefetch that hits in the DTLB does not impact the L2-TLB MPKI [32], [33]. Another interesting observation is that SEMPRE is more effective at TLB prefetching than both IMP and SPP. The main takeaway is that SEMPRE significantly reduces TLB misses because it prefetches for memory access patterns that would miss in the TLB in the absence of SEMPRE.

B. Analyzing SEMPRE's Benefits

To explain where SEMPRE's gains originate, Figure 12 presents the coverage, timeliness, accuracy, and useless prefetch traffic for SEMPRE and the best-performing prefetcher from previous sections, SPP.

1) Coverage and Timeliness: Figure 12 (a) classifies every demand miss as *covered-timely* (the prefetched line is present when the demand arrives), *covered-late* (the prefetch is still

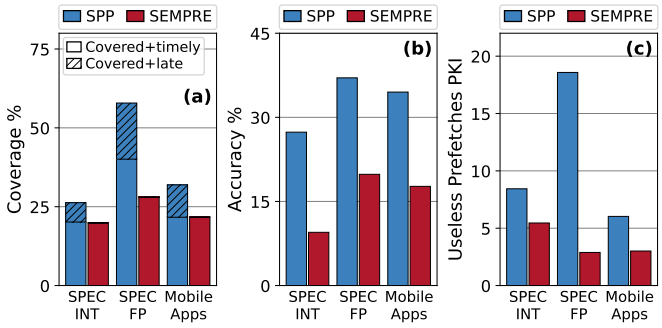

Fig. 12: Impact on coverage, accuracy, and useless prefetches per kilo instructions (uPKI).

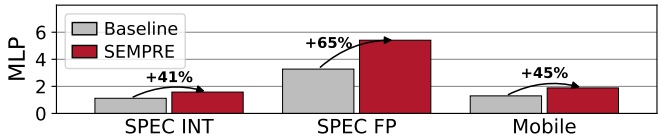

Fig. 13: Impact on Memory Level Parallelism (MLP).

in flight when demand arrives), or *not covered*. SEMPRE eliminates 19.9%, 28.1%, and 21.8% of demand misses on the SPEC INT, SPEC FP, and mobile applications and this coverage is overwhelmingly timely because the OoH scheme issues prefetches many iterations ahead of the consuming load. For example, on mobile, 99.9% of its prefetches that cover a demand miss are timely. SEMPRE's coverage is narrow by design: it builds slices *only* for performance-critical flaky loads—the irregular, long-history accesses that prior prefetchers cannot capture—and is intended to compose with a spatial prefetcher that handles the regular patterns. Indeed, the SEMPRE+SPP configuration in Figure 10 makes this explicit: SPP covers the spatial misses, while SEMPRE contributes the residual misses that no prior prefetcher captures.

2) High-Value Coverage: SEMPRE covers misses that are critical for performance, which explains why its low coverage provides significant gains in Figure 10. These are the accesses that miss all the way to DRAM and that serialize the instruction window, so removing even a few of them provides great benefits. Two observations support this. Figure 13 shows that SEMPRE raises Memory Level Parallelism (MLP), *i.e.*, the average number of concurrently outstanding memory accesses. Indeed, SEMPRE raises MLP by 41%, 65%, and 45% on SPEC INT, SPEC FP, and mobile applications, respectively, indicating that previously serialized loads are now overlapped. Second, we measured that the per-covered-miss performance contribution is much higher for SEMPRE than any other prefetcher, since not coverage volume but performance recovered per covered miss is what brings the highest gains.

3) Accuracy and Prefetch Traffic: The OoH scheme of SEMPRE speculates aggressively: it anchors slices on value recurrence, and launches prefetches many iterations ahead, so a large fraction of the addresses it generates are ultimately not used. Its prefetch accuracy is 9.5%, 19.8%, and 17.7% on SPEC INT, SPEC FP, and mobile applications, respectively, as shown in Figure 12 (b). Critically, this low accuracy does

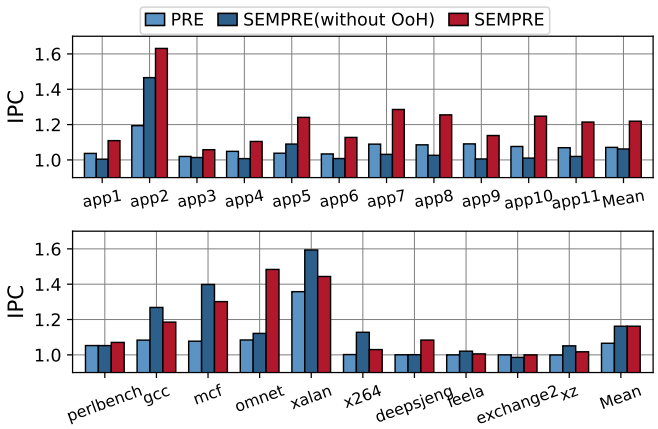

Fig. 14: IPC comparison between SEMPRE, Precise Runahead (PRE) [23], and SEMPRE without the OoH scheme.

not translate into wasted bandwidth, because SEMPRE issues very few prefetches: injection is gated to performance-critical flaky loads, and the adaptive throttle (Section IV-D and Section IV-E2) disables slices whose prefetches go unused before they accumulate. The result is that absolute useless prefetch traffic (lines fetched but evicted before use) is the lowest of the aggressive prefetchers evaluated. Specifically, Figure 12 (c) shows that SEMPRE issues 3.0 useless prefetches per kilo instructions (uPKI) on mobile versus 6.0 for SPP, and 2.9 versus 19.0 on SPEC FP. In other words, SEMPRE trades speculation accuracy for tightly bounded waste: it explores aggressively to reach the long-history, irregular accesses no other prefetcher covers, while the validation-and-throttle pipeline ensures the cost of incorrect speculation is paid in discarded slices rather than evicted cache lines.

Overall, SEMPRE's gains come not from covering more misses but from covering the performance-critical ones. The misses it eliminates are the MLP-blocking accesses that no prior prefetcher reaches, and it eliminates them on time and with little wasted bandwidth. Pairing it with a spatial prefetcher such as SPP recovers the regular accesses and raises the performance gains further, as shown in Figure 10.

C. Comparison with Runahead Techniques

Runahead schemes [20]–[22], [46], [47] speculatively execute code ahead of time. The state-of-the-art slice-based runahead engines are VR [24], DVR [25], and PRE [23]. The limitations of prior runahead schemes and how SEMPRE addresses them were discussed in Section II. This section compares SEMPRE with PRE [23] since it has fewer limiting factors than VR and DVR (Table I). Figure 14 presents the performance comparison across all considered workloads.

SEMPRE outperforms PRE by 9.6% and 14.8% for the SPEC INT and the mobile applications, respectively, because PRE 1) relies on continued execution of the same program context past memory stalls and manages a complicated speculative state for that purpose, 2) does not modify the executed code but filters out code not required for memory address calculation, 3) does no extrapolation and is limited in range to

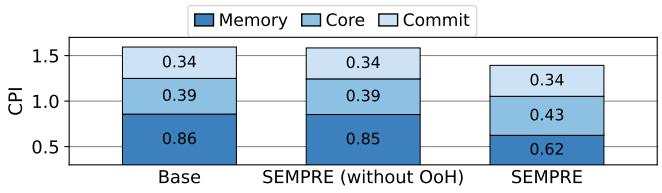

Fig. 15: CPI stall-stack for the mobile applications.

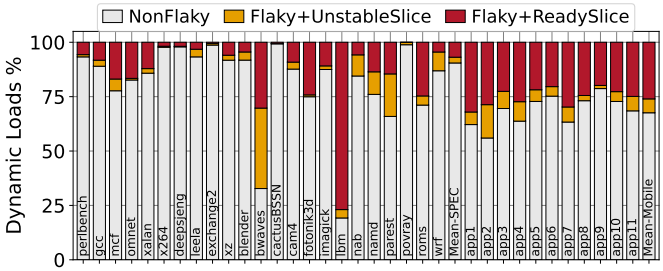

Fig. 16: Breakdown of dynamic loads.

what could fit in the enhanced out-of-order window, thus being constrained by finite out-of-order depths. On the other hand, SEMPRE i) can extrapolate across any level of nesting by picking the correct stride to leverage, ii) can peek into future iso-context iterations without having to execute everything in the middle, iii) builds scalar slices that adapt well to irregularities in code or address patterns, and iv) creates approximate slices for operations with massive history tracking requirements that cannot be captured by prior schemes (Section IV-C). The SPEC FP behave similarly to SPEC INT.

D. Importance of Out-of-History Scheme

This section quantifies the benefits of the OoH scheme (Section IV-C) by comparing SEMPRE with a version of SEMPRE without OoH. Figure 14 shows that SEMPRE outperforms SEMPRE(without OoH) for the mobile applications because most of these applications require massive history lengths, as shown in Figure 5, that cannot be captured by realistic History Queue sizes. Indeed, SEMPRE(without OoH) provides almost zero speedup for most of the mobile applications since it is unable to build slices for most flaky loads due to limited History Queue size. The OoH scheme manages to build slices even for loads requiring very large histories by exploiting the uop data for approximate slice creation (Section IV-C). Therefore, we conclude that the OoH scheme is an essential part of SEMPRE that delivers large gains by building approximate slices without having the entire retired code flow history available.

To attribute these gains to cycles, Figure 15 presents a CPI stall-stack for the mobile applications under three configurations: the baseline, SEMPRE without OoH, and SEMPRE. Relative to the baseline (CPI 1.59), SEMPRE reduces CPI to 1.39, and the reduction comes from memory-stall cycles, which fall from 0.86 to 0.62 CPI; core-stall cycles rise slightly (0.39 to 0.43) as the SEE executes slices and previously hidden core stalls are exposed, and committing cycles are unchanged, for a net -0.20 CPI. SEMPRE (without OoH) leaves CPI at

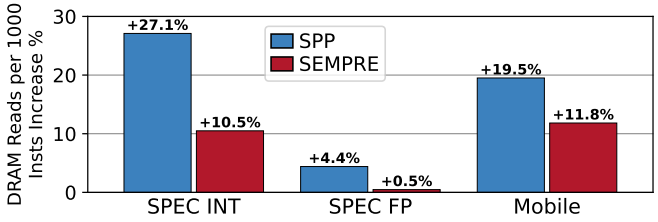

Fig. 17: Impact on DRAM bandwidth.

Suite	NonFlaky	Flaky+UnstableSlice	Flaky+ReadySlice
SPEC INT	222.3	7.9	18.5
SPEC FP	144.5	92.4	86.7
Mobile	234.4	3.6	11.7

TABLE IV: Flaky load frequency by type per 1K instructions.

1.58 with memory-stall cycles essentially unchanged from the baseline (0.86 to 0.85), with the only exception of `app2`, which needs histories that mainly fit in the History Queue. Without OoH, SEMPRE cannot build slices for the long-history loads that dominate the mobile applications (Figure 5), confirming that SEMPRE's gains on the mobile applications come specifically from the memory-stall cycles removed by the long-history loads OoH unlocks, consistent with the near-zero gain of SEMPRE (without OoH) in Figure 14.

The OoH scheme is seldom beneficial for SPEC INT. It improves IPC in benchmarks like `omnet` and `deepsjeng` but hurts performance in `gcc`, `mcf`, and `xalan` because it builds approximate slices when there is not enough retired code history. The per-slice confidence throttle (Section IV-E2) discards these low-accuracy OoH slices once detected, bounding their cost; the residual loss equals the gap between SEMPRE and SEMPRE-without-OoH on these benchmarks in Figure 14. A per-context gate that suppresses OoH anchoring once a context's slices prove low-confidence would reduce this residual while preserving the gains on the mobile applications.

E. Additional Studies

1) Slice Characterization: Figure 16 analyzes the properties of dynamic loads by breaking them down in: (i) non-flaky loads, *i.e.*, loads not deemed hot enough or not causing enough misses to be interesting for creating a slice, (ii) flaky loads with unstable slices, *i.e.*, loads for which SEMPRE builds slices but the slices turned out to be unstable (❸ and ❽ in Figure 8), and (iii) flaky loads with ready slices to be used for prefetching (❼ in Figure 8). Figure 16 shows that the majority of loads are not classified as flaky since they did not exhibit high occurrence and cache miss rates (NonFlaky category). Regarding the flaky loads, we observe that the majority of the slices generated by SEMPRE for these loads are ready slices, thus being used to generate prefetches (Flaky+ReadySlice category). However, there is a fraction of flaky loads with unstable slices (Flaky+UnstableSlice category) because some flaky loads do not produce consistent slices, making SEMPRE abort these slices and not use them (Section IV-D).

To make the opportunity explicit, Table IV reports flaky load frequency per 1K instructions; these are normalized rates

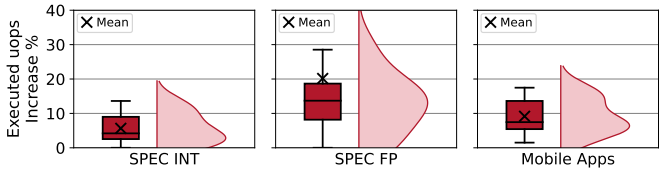

Fig. 18: Impact of SEMPRE on executed uops.

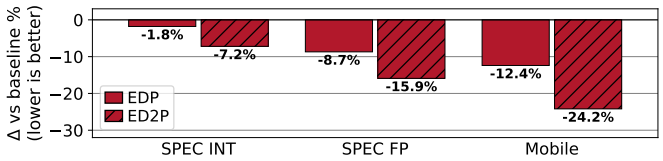

Fig. 19: Impact of SEMPRE on EDP and ED2P.

aggregated per workload suite. Flaky loads with ready slices, *i.e.*, those for which SEMPRE builds usable slices, occur at 18.5, 86.7, and 11.7 per 1K instructions on the SPEC INT, SPEC FP, and mobile applications, respectively, while unstable flaky loads add 7.9, 92.4, and 3.6 per 1K instructions. SPEC FP benchmarks have the highest frequency, reflecting streaming repetition of a few loads, whereas the mobile applications have a comparable-or-lower flaky-load frequency spread across an order of magnitude more unique slices (Figure 4); a single load reached through different control-flow contexts yields several slices (Section IV-A). This is why mobile applications require a larger Slice Array, as Table V shows.

2) Impact on DRAM Bandwidth: Our evaluation uses LPDDR5 at 5500 MTPS (Table II). To quantify the impact on bandwidth, we measure DRAM read traffic per 1K committed instructions for the baseline, SPP (best-performing prior prefetcher in Figure 10), and SEMPRE. Figure 17 shows that SEMPRE's added read traffic is below SPP. For example, on the mobile applications, SEMPRE adds 11.8% over the baseline versus 19.5% for SPP. Critically, SEMPRE achieves this with lower wasted traffic than SPP—3.0 useless prefetched lines on mobile versus 6.0 for SPP, as shown in Section V-B and Figure 12 (c)—because prefetch injection is gated to only performance-critical loads and non-proven useful slices are throttled (Section IV-D, Section IV-E2). In absolute terms, the mobile applications consume 0.92GB/s, ∼2.1% (per-app 0.8%–3.1%) of the 44GB/s LPDDR5 peak; even with SEMPRE's added traffic, the workloads remain at ∼2.4% of peak, confirming they are latency-bound rather than bandwidth-bound. Consistent with this, reducing DRAM to 3200 MTPS yields a negligible performance loss.

3) Energy and Power: Figure 18 quantifies SEMPRE's activity overhead: it executes 5.7%, 20.1%, and 9.1% more uops than the baseline on SPEC INT, SPEC FP, and the mobile applications, respectively. These additional uops are SEE slice uops issued for prefetching; since the SEE has no front-end and no caches and runs as a half-clock, single-issue engine, each costs well below a main-core uop. To turn this activity into energy, we build a per-1K-instruction activity profile (committed uops, SEE uops, L1/L2/LLC accesses, and DRAM traffic) and apply an activity-based energy model with on-chip per-event energies from CACTI [48]. The added dynamic energy is dominated by the SEE slice uops, together with the additional accesses quantified in Section V-E2. Because SEMPRE trades activity for latency, raw IPC/watt is approximately flat: the IPC gain is offset by the added power. However, it is not the right metric for a latency mechanism, as it does not credit the reduced runtime. Figure 19 presents the energy-delay results. Overall, SEMPRE improves on every benchmark suite, reducing EDP by 1.8%/8.7%/12.4% and ED^{2}P by 7.2%/15.9%/24.2% on SPEC INT/FP/mobile, respectively.

Component	Description	Storage (KB)
Flaky Load Table	64-set, 4-way, pLRU	1.2KB
History Queue	256-entry FIFO (128 bits/entry)	4.0KB
Rename Cache	16-entry stack buffer	1.0KB
Walkers	64-bit temp registers, 32-bit bitmap	1.2KB
Slice Array (spec/mobile)	256/4096-entry, direct mapped	2.2KB/36.0KB
UDQ	64-set, 4-way, 1/1 read/write port	2.0KB
Slice Execution Engine	1 ALU, 8 registers	0.8KB
Update Queue	128-entry FIFO circular buffer	1.0KB
Total Size (spec/mobile)		13.4KB/47.2KB

TABLE V: Storage overhead of SEMPRE.

4) Storage and Area Overheads: Table V details SEMPRE's storage overhead. The Slice Array has more entries for the mobile applications than SPEC, as the former have more unique code slices. The History Queue has 4 banks; each bank has one read port since it is used by one Walker and two write ports (empirically selected since SEMPRE targets memory-bound applications). We convert SEMPRE's storage in Table V to area using per-bit SRAM area from CACTI [48]. The logic (Walker FSMs, SEE (a 2-stage single-issue datapath with one ALU and eight registers, Section IV-E)) is small relative to the SRAM and is estimated from the datapath/standard-cell area factors in the target core's physical design library. Overall, the area overhead of SEMPRE is ∼2% of the simulated core.

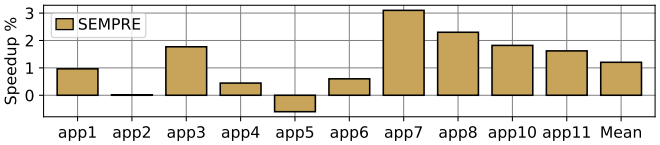

Fig. 20: Evaluation using an in-house simulator.

5) Industrial Simulator: Figure 20 quantifies the speedups of SEMPRE on an in-house industrial simulator using the mobile workloads (app9 is excluded due to incomplete simulation) over a baseline with multiple spatial, temporal, and indirect prefetchers and schemes that dynamically throttle prefetching while optimizing training across execution phases. The main takeaway of this study is that SEMPRE offers gains even on a highly tuned industrial simulation framework with multiple prefetchers and coordination schemes.

VI. Related Work

Exploiting Code Slices. SVR [49] extends vector runahead to scalar and irregular access on energy-efficient cores; like VR/DVR it targets dependent-load chains reachable from the stall point, whereas SEMPRE targets recurrences separated by histories beyond any reachable window. Zilles et al. [50]

use slices to run ahead along predicted paths, aiding hard-to-predict branches. Carlson et al. [30] dynamically learn load dependence chains to expedite execution. Helper threads [51] and slice processors [52] also perform slice-based prefetching, and Hardware Scouting [53] runs ahead on separate threads. DSP [54] uses helper threads for delinquent loads but is limited by a narrow, transient history window and lacks long-range dependence tracking. SEMPRE constructs validated slices from retired instructions, ensuring stability, reusability, and long-range dependence handling via OoH. DGP [55] uses profiling and compiler support to execute memory access instructions. Jamilan et al. [56] propose a profile-guided software prefetcher that co-optimizes prefetch distance and injection via an analytical model. CRISP [27] is a software scheme that uses profile-guided analysis to identify and execute slices for critical accesses. SEMPRE differs in when and how slices are built: CRISP uses slices from prior executions, whereas SEMPRE constructs them at runtime, adapting to input-dependent and unseen patterns. SEMPRE uses long-range dependence prediction, enabling approximate slice construction when code history is insufficient—which, to our knowledge, prior slice-based approaches do not provide—and selectively builds and refines slices online, scaling to large code footprints and complex control flow. Prior work [57] introduces an offline, profile-guided taxonomy correlating patterns with dataflow. SEMPRE and [27], [57] are orthogonal: SEMPRE is hardware engine while [27], [57] are profile-guided software prefetching; a scheme could use [57] to select hot regions and apply SEMPRE for irregular kernels. This line of work also adds deployment complexity and hinders legacy-code support. CFP [58] buffers a miss's dependent instructions and re-executes them when the miss returns, reclaiming pipeline resources rather than prefetching. SEMPRE instead builds a validated address-generating slice from retired history and reuses it across future instances, with value-based approximate slice creation when history is unavailable.

Slice Generation. SEMPRE's model for slice construction (Section IV-B) builds on IBDA's dependence tracking backbone [30] but extends it. Unlike IBDA, which infers only immediate register-based strides, SEMPRE extrapolates stable stride patterns for deeper cross-iteration prefetching. IBDA tracks dependences only through register chains and loses correlations on register spills; SEMPRE follows store→load relationships to reconnect spill pairs and capture address computations propagating through memory. IBDA flattens loops via iteration repetition, whereas SEMPRE represents loop nesting for multi-dimensional extrapolation. IBDA assumes static slice structures once found, while SEMPRE repeatedly validates slices at runtime for stability under control or dependence changes. IBDA derives slice relationships from in-flight analysis, whereas SEMPRE collects information at retirement, exposing constants and strides that transient tracking cannot observe. Unlike IBDA's reliance on explicit address-dependence chains, SEMPRE adds value-based speculation, propagating recurring values across distant iterations even when full dependence chains are unavailable.

Cache Prefetching. Some prefetchers [7], [41], [59], [60] capture patterns through common recurring strides. Address-correlating prefetchers [5], [6], [10], [11], [16]–[18], [26], [61], [62] exploit past temporal locality to predict future accesses. Spatial prefetchers [3], [4], [8], [12], [13], [40], [63]–[65] seek recurring spatial patterns. Irregular prefetchers [15] handle data lacking spatio-temporal locality. ISB [66] and MISB [67] restructure the dataset spatially. B-Fetch [68] and SB-Fetch [69] use branch history to detect strides in registers used for address generation. DX100 [70] accelerates bulk indirect accesses through a compiler-managed offload interface, whereas SEMPRE is a hardware engine without software support.

Cache prefetching also interacts with address translation. Prior work (i) proposes a scheme that makes lower-level cache prefetchers aware of page size information [33] and (ii) shows that page-cross prefetches at VIPT caches should be issued selectively and proposes a dedicated page-cross prefetch filter for prefetch engines operating with virtual addresses [32]. SEMPRE instead crosses pages as a by-product of following dependence chains, prefetching translations in the TLB hierarchy without using a dedicated TLB prefetch engine.

TLB Prefetching and Optimizations. Early schemes exploit recency and inter-reference distances in the page reference stream [71], [72], later work shares translations across cores [73], and recent art proposes composite TLB prefetching schemes for data and code translations [44], [45]. Another direction manages TLB and cache contents jointly rather than prefetching translations, co-designing their replacement policies [74], [75]. These approaches are complementary to SEMPRE since it employs no dedicated TLB engine and no auxiliary TLB prefetch buffer: the prefetched translations originate from SEMPRE's prefetch requests that cross page boundaries, and are stored directly into the TLB hierarchy. A dedicated TLB prefetcher or a specialized TLB-cache management policy can therefore compose with SEMPRE.

VII. Conclusions

Slice-based microarchitectural schemes share a common constraint: they work only when the dependence chain of interest fits in the history the hardware can afford to track. This work demonstrates that this limitation can be addressed by combining semantic locality with long-range dependence prediction based on value equality, as it enables approximate slice construction beyond practical history limits.

In the hardware prefetching domain, we realize this concept in *SEMantic PREfetcher (SEMPRE)*, an engine that builds slices only for performance-critical memory operations and executes them to prefetch access patterns that exhibit no spatio-temporal locality. SEMPRE outperforms previously proposed cache prefetchers and runahead schemes across SPEC CPU and contemporary mobile applications, reduces demand TLB misses by prefetching across virtual page boundaries, and composes with spatial prefetchers for further gains. More broadly, approximate slice construction is applicable to any slice-based microarchitectural technique for which the requisite dependence history exceeds realistic tracking capacity.

References

[1] Chips and Cheese, "Hot chips 2023: Arm's Neoverse V2," https://chipsandcheese.com/2023/09/11/hot-chips-2023-arms-neoverse-v2/, 2023, accessed: 2026-08-24.

[2] B. Falsafi and T. F. Wenisch, "A primer on hardware prefetching," *Synthesis Lectures on Computer Architecture*, vol. 9, no. 1, 2014. [Online]. Available: https://doi.org/10.1007/978-3-031-01743-8

[3] M. Bakhshalipour, M. Shakerinava, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Bingo spatial data prefetcher," in *Proceedings of the 25th International Symposium on High Performance Computer Architecture*, ser. HPCA '19. IEEE, 2019, pp. 399–411. [Online]. Available: https://doi.org/10.1109/HPCA.2019.00053

[4] A. Navarro-Torres, B. Panda, J. Alastruey-Benedé, P. Ibáñez, V. Viñals-Yúfera, and A. Ros, "Berti: An accurate local-delta data prefetcher," in *Proceedings of the 55th International Symposium on Microarchitecture*, ser. MICRO '22. IEEE, 2022, pp. 975–991. [Online]. Available: https://doi.org/10.1109/MICRO56248.2022.00072

[5] H. Wu, K. Nathella, J. Pusdesris, D. Sunwoo, A. Jain, and C. Lin, "Temporal prefetching without the off-chip metadata," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: Association for Computing Machinery, 2019, pp. 996–1008. [Online]. Available: https://doi.org/10.1145/3352460.3358300

[6] S. Ainsworth and L. Mukhanov, "Triangel: A high-performance, accurate, timely on-chip temporal prefetcher," in *Proceedings of the 51st Annual International Symposium on Computer Architecture*, ser. ISCA '24. Los Alamitos, CA, USA: IEEE Computer Society, 2024, pp. 1202–1216. [Online]. Available: https://doi.org/10.1109/ISCA59077.2024.00090

[7] P. Michaud, "Best-offset hardware prefetching," in *Proceedings of the 22nd International Symposium on High Performance Computer Architecture*, ser. HPCA '16. IEEE, 2016, pp. 469–480. [Online]. Available: https://doi.org/10.1109/HPCA.2016.7446087

[8] S. Somogyi, T. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Spatial memory streaming," in *Proceedings of the 33rd International Symposium on Computer Architecture*, ser. ISCA '06. IEEE, 2006, pp. 252–263. [Online]. Available: https://doi.org/10.1109/ISCA.2006.38

[9] T. F. Wenisch, M. Ferdman, A. Ailamaki, B. Falsafi, and A. Moshovos, "Practical off-chip meta-data for temporal memory streaming," in *Proceedings of the 15th International Symposium on High Performance Computer Architecture*, ser. HPCA '09. IEEE, 2009, pp. 79–90. [Online]. Available: https://doi.org/10.1109/HPCA.2009.4798239

[10] M. Bakhshalipour, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Domino temporal data prefetcher," in *Proceedings of the 24th International Symposium on High Performance Computer Architecture*, ser. HPCA '18. IEEE, 2018, pp. 131–142. [Online]. Available: https://doi.org/10.1109/HPCA.2018.00021

[11] K. Nesbit and J. Smith, "Data cache prefetching using a global history buffer," in *Proceedings of the 10th International Symposium on High Performance Computer Architecture*, ser. HPCA '04. IEEE, 2004, pp. 96–105. [Online]. Available: https://doi.org/10.1109/HPCA.2004.10030

[12] G. Gerogiannis and J. Torrellas, "Micro-armed bandit: Lightweight & reusable reinforcement learning for microarchitecture decision-making," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '23. New York, NY, USA: Association for Computing Machinery, 2023, pp. 698–713. [Online]. Available: https://doi.org/10.1145/3613424.3623780

[13] C. Block, G. Gerogiannis, and J. Torrellas, "Micro-MAMA: Multi-agent reinforcement learning for multicore prefetching," in *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '25. New York, NY, USA: Association for Computing Machinery, 2025, pp. 884–898. [Online]. Available: https://doi.org/10.1145/3725843.3756096

[14] M. Cavus, R. Sendag, and J. J. Yi, "Informed prefetching for indirect memory accesses," *ACM Transactions on Architecture and Code Optimization*, vol. 17, no. 1, pp. 4:1–4:29, 2020. [Online]. Available: https://doi.org/10.1145/3374216

[15] F. Xue, C. Han, X. Li, J. Wu, T. Zhang, T. Liu, Y. Hao, Z. Du, Q. Guo, and F. Zhang, "Tyche: An efficient and general prefetcher for indirect memory accesses," *ACM Transactions on Architecture and Code Optimization*, vol. 21, no. 2, pp. 30:1–30:26, 2024. [Online]. Available: https://doi.org/10.1145/3641853

[16] A. Roth, A. Moshovos, and G. S. Sohi, "Dependence based prefetching for linked data structures," in *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '98. New York, NY, USA: Association for Computing Machinery, 1998, pp. 115–126. [Online]. Available: https://doi.org/10.1145/291069.291034

[17] A. Roth and G. S. Sohi, "Effective jump-pointer prefetching for linked data structures," in *Proceedings of the 26th International Symposium on Computer Architecture*, ser. ISCA '99. USA: IEEE Computer Society, 1999, pp. 111–121. [Online]. Available: https://doi.org/10.1145/307338.300989

[18] M. Bekerman, S. Jourdan, R. Ronen, G. Kirshenboim, L. Rappoport, A. Yoaz, and U. Weiser, "Correlated load-address predictors," in *Proceedings of the 26th International Symposium on Computer Architecture*, ser. ISCA '99. New York, NY, USA: Association for Computing Machinery, 1999, pp. 54–63. [Online]. Available: https://doi.org/10.1145/307338.300984

[19] X. Yu, C. J. Hughes, N. Satish, and S. Devadas, "IMP: Indirect memory prefetcher," in *Proceedings of the 48th International Symposium on Microarchitecture*, ser. MICRO '15. New York, NY, USA: Association for Computing Machinery, 2015, pp. 178–190. [Online]. Available: https://doi.org/10.1145/2830772.2830807

[20] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt, "Runahead execution: an alternative to very large instruction windows for out-of-order processors," in *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, ser. HPCA '03. IEEE, 2003, pp. 129–140. [Online]. Available: https://doi.org/10.1109/HPCA.2003.1183532

[21] M. Hashemi and Y. N. Patt, "Filtered runahead execution with a runahead buffer," in *Proceedings of the 48th International Symposium on Microarchitecture*, ser. MICRO '15. New York, NY, USA: Association for Computing Machinery, 2015, pp. 358–369. [Online]. Available: https://doi.org/10.1145/2830772.2830812

[22] M. Hashemi, O. Mutlu, and Y. N. Patt, "Continuous runahead: Transparent hardware acceleration for memory intensive workloads," in *Proceedings of the 49th International Symposium on Microarchitecture*, ser. MICRO-49. Piscataway, NJ, USA: IEEE, 2016, pp. 1–12. [Online]. Available: https://doi.org/10.1109/MICRO.2016.7783764

[23] A. Naithani, J. Feliu, A. Adileh, and L. Eeckhout, "Precise runahead execution," in *Proceedings of the 26th International Symposium on High Performance Computer Architecture*, ser. HPCA '20. IEEE, 2020, pp. 397–410. [Online]. Available: https://doi.org/10.1109/HPCA47549.2020.00040

[24] A. Naithani, S. Ainsworth, T. M. Jones, and L. Eeckhout, "Vector runahead," in *Proceedings of the 48th Annual International Symposium on Computer Architecture*, ser. ISCA '21. IEEE, 2021, pp. 195–208. [Online]. Available: https://doi.org/10.1109/ISCA52012.2021.00024

[25] A. Naithani, J. Roelandts, S. Ainsworth, T. M. Jones, and L. Eeckhout, "Decoupled vector runahead," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '23. New York, NY, USA: Association for Computing Machinery, 2023, pp. 17–31. [Online]. Available: https://doi.org/10.1145/3613424.3614255

[26] L. Peled, S. Mannor, U. Weiser, and Y. Etsion, "Semantic Locality and Context-Based Prefetching Using Reinforcement Learning," in *Proceedings of the 42nd International Symposium on Computer Architecture*, ser. ISCA '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 285–297. [Online]. Available: https://doi.org/10.1145/2749469.2749473

[27] H. Litz, G. Ayers, and P. Ranganathan, "CRISP: Critical slice prefetching," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '22. New York, NY, USA: Association for Computing Machinery, 2022, pp. 300–313. [Online]. Available: https://doi.org/10.1145/3503222.3507745

[28] G. Vavouliotis, T. Rollet, D. B. Bartolini, B. Grot, L. Peled, and L. Yang, "Bumper: Hinting instruction usefulness for robust unified caches," in *Proceedings of the 53rd Annual International Symposium on Computer Architecture*, ser. ISCA '26, 2026, pp. 2029–2045. [Online]. Available: https://doi.org/10.1109/ISCA66397.2026.00145

[29] Standard Performance Evaluation Corporation, "SPEC CPU 2017," https://www.spec.org/cpu2017/, 2017, accessed: 2026-08-24.

[30] T. E. Carlson, W. Heirman, O. Allam, S. Kaxiras, and L. Eeckhout, "The load slice core microarchitecture," in *Proceedings of the 42nd*

Annual International Symposium on Computer Architecture, ser. ISCA '15. New York, NY, USA: ACM, 2015, pp. 272–284. [Online]. Available: https://doi.org/10.1145/2749469.2750407

[31] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011. [Online]. Available: https://doi.org/10.1145/2024716.2024718

[32] G. Vavouliotis, M. Torrents, B. Grot, K. Kalaitzidis, L. Peled, and M. Casas, "To cross, or not to cross pages for prefetching?" in *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, ser. HPCA '25. IEEE, 2025, pp. 188–203. [Online]. Available: https://doi.org/10.1109/HPCA61900.2025.00025

[33] G. Vavouliotis, G. Chacon, L. Alvarez, P. V. Gratz, D. A. Jiménez, and M. Casas, "Page size aware cache prefetching," in *Proceedings of the 55th International Symposium on Microarchitecture*, ser. MICRO '22. IEEE, 2022, pp. 956–974. [Online]. Available: https://doi.org/10.1109/MICRO56248.2022.00070

[34] S. Beamer, K. Asanovic, and D. A. Patterson, "The GAP benchmark suite," arXiv:1508.03619 [cs.DC], 2015. [Online]. Available: http://arxiv.org/abs/1508.03619

[35] A. V. Jamet, G. Vavouliotis, D. A. Jiménez, L. Alvarez, and M. Casas, "Practically tackling memory bottlenecks of graph-processing workloads," in *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2024, pp. 1034–1045. [Online]. Available: https://doi.org/10.1109/IPDPS57955.2024.00096

[36] G. S. Tyson and T. M. Austin, "Improving the accuracy and performance of memory communication through renaming," in *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, ser. MICRO 30. Washington, DC, USA: IEEE Computer Society, 1997, pp. 218–227. [Online]. Available: https://doi.org/10.1109/MICRO.1997.645812

[37] D. Patsidis and G. Vavouliotis, "Context-aware set dueling for dynamic policy arbitration," *IEEE Computer Architecture Letters*, vol. 24, no. 2, pp. 301–304, 2025. [Online]. Available: https://doi.org/10.1109/LCA.2025.3617159

[38] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder, "Using SimPoint for accurate and efficient simulation," in *Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '03. New York, NY, USA: Association for Computing Machinery, 2003, pp. 318–319. [Online]. Available: https://doi.org/10.1145/781027.781076

[39] M. Carpen-Amarie, G. Vavouliotis, K. Tovletoglou, B. Grot, and R. Mueller, "Concurrent GCs and modern java workloads: A cache perspective," in *Proceedings of the 2023 ACM SIGPLAN International Symposium on Memory Management*, ser. ISMM 2023. New York, NY, USA: Association for Computing Machinery, 2023, pp. 71–84. [Online]. Available: https://doi.org/10.1145/3591195.3595269

[40] J. Kim, S. H. Pugsley, P. V. Gratz, A. L. N. Reddy, C. Wilkerson, and Z. Chishti, "Path confidence based lookahead prefetching," in *Proceedings of the 49th International Symposium on Microarchitecture*, ser. MICRO '16. IEEE, 2016, pp. 1–12. [Online]. Available: https://doi.org/10.1109/MICRO.2016.7783763

[41] Y. Ishii, M. Inaba, and K. Hiraki, "Access map pattern matching for data cache prefetch," in *Proceedings of the 23rd International Conference on Supercomputing*, ser. ICS '09. New York, NY, USA: Association for Computing Machinery, 2009, pp. 499–500. [Online]. Available: https://doi.org/10.1145/1542275.1542349

[42] Arm Ltd., "Arm Cortex-A78AE core technical reference manual, revision r0p1: CPUECTLR_EL1, CPU extended control register, EL1," https://developer.arm.com/documentation/101779/0001/Register-descriptions/AArch64-system-registers/CPUECTLR-EL1--CPU-Extended-Control-Register--EL1, 2020, accessed: 2026-08-24.

[43] ——, "Arm Neoverse N2: Arm's 2nd generation high performance infrastructure CPUs and system IPs," https://hc33.hotchips.org/assets/program/conference/day1/20210818_Hotchips_NeoverseN2.pdf, 2021, hot Chips 33 presentation. Accessed: 2026-08-24.

[44] G. Vavouliotis, L. Alvarez, B. Grot, D. Jiménez, and M. Casas, "Morrigan: A composite instruction TLB prefetcher," in *Proceedings of the 54th International Symposium on Microarchitecture*, ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, pp. 1138–1153. [Online]. Available: https://doi.org/10.1145/3466752.3480049

[45] G. Vavouliotis, L. Alvarez, V. Karakostas, K. Nikas, N. Koziris, D. A. Jiménez, and M. Casas, "Exploiting page table locality for agile TLB prefetching," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, ser. ISCA '21. IEEE, 2021, pp. 85–98. [Online]. Available: https://doi.org/10.1109/ISCA52012.2021.00016

[46] J. D. Collins, H. Wang, D. M. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J. P. Shen, "Speculative precomputation: Long-range prefetching of delinquent loads," in *Proceedings of the 28th Annual International Symposium on Computer Architecture*. IEEE, 2001, pp. 14–25. [Online]. Available: https://doi.org/10.1109/isca.2001.937427

[47] I. Atta, X. Tong, V. Srinivasan, I. Baldini, and A. Moshovos, "Self-contained, accurate precomputation prefetching," in *Proceedings of the 48th International Symposium on Microarchitecture*, ser. MICRO-48. New York, NY, USA: ACM, 2015, pp. 153–165. [Online]. Available: https://doi.org/10.1145/2830772.2830816

[48] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "CACTI 7: New tools for interconnect exploration in innovative off-chip memories," *ACM Transactions on Architecture and Code Optimization*, vol. 14, no. 2, pp. 14:1–14:25, 2017. [Online]. Available: https://doi.org/10.1145/3085572

[49] J. Roelandts, A. Naithani, S. Ainsworth, T. M. Jones, and L. Eeckhout, "Scalar vector runahead," in *Proceedings of the 57th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '24. IEEE, 2024, pp. 1367–1381. [Online]. Available: https://doi.org/10.1109/MICRO61859.2024.00101

[50] C. Zilles and G. Sohi, "Execution-based prediction using speculative slices," in *Proceedings of the 28th Annual International Symposium on Computer Architecture*, ser. ISCA '01. New York, NY, USA: ACM, 2001, pp. 2–13. [Online]. Available: https://doi.org/10.1145/379240.379246

[51] P. Xekalakis, N. Ioannou, and M. Cintra, "Combining thread level speculation helper threads and runahead execution," in *Proceedings of the 23rd International Conference on Supercomputing*, ser. ICS '09. New York, NY, USA: ACM, 2009, pp. 410–420. [Online]. Available: https://doi.org/10.1145/1542275.1542333

[52] A. Moshovos, D. N. Pnevmatikatos, and A. Baniasadi, "Slice-processors: An implementation of operation-based prediction," in *Proceedings of the 15th International Conference on Supercomputing*, ser. ICS '01. New York, NY, USA: ACM, 2001, pp. 321–334. [Online]. Available: https://doi.org/10.1145/377792.377856

[53] S. Chaudhry, P. Caprioli, S. Yip, and M. Tremblay, "High-performance throughput computing," *IEEE Micro*, vol. 25, no. 3, pp. 32–45, 2005. [Online]. Available: https://doi.org/10.1109/MM.2005.49

[54] J. Collins, D. Tullsen, H. Wang, and J. Shen, "Dynamic speculative precomputation," in *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, ser. MICRO-34, 2001, pp. 306–317. [Online]. Available: https://doi.org/10.1109/MICRO.2001.991128

[55] M. Annavaram, J. Patel, and E. Davidson, "Data prefetching by dependence graph precomputation," in *Proceedings of the 28th Annual International Symposium on Computer Architecture*, ser. ISCA '01, 2001, pp. 52–61. [Online]. Available: https://doi.org/10.1109/ISCA.2001.937432

[56] S. Jamilan, T. A. Khan, G. Ayers, B. Kasikci, and H. Litz, "Apt-get: profile-guided timely software prefetching," in *Proceedings of the Seventeenth European Conference on Computer Systems*, ser. EuroSys '22. New York, NY, USA: Association for Computing Machinery, 2022, pp. 747–764. [Online]. Available: https://doi.org/10.1145/3492321.3519583

[57] G. Ayers, H. Litz, C. Kozyrakis, and P. Ranganathan, "Classifying memory access patterns for prefetching," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 513–526. [Online]. Available: https://doi.org/10.1145/3373376.3378498

[58] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton, "Continual flow pipelines," in *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '04. New York, NY, USA: ACM, 2004, pp. 107–119. [Online]. Available: https://doi.org/10.1145/1024393.1024407

[59] S. H. Pugsley, Z. Chishti, C. Wilkerson, P.-f. Chuang, R. L. Scott, A. Jaleel, S.-L. Lu, K. Chow, and R. Balasubramonian, "Sandbox prefetching: Safe run-time evaluation of aggressive prefetchers," in

Proceedings of the 20th International Symposium on High Performance Computer Architecture, ser. HPCA '14, 2014, pp. 626–637. [Online]. Available: https://doi.org/10.1109/HPCA.2014.6835971

[60] M. Shevgoor, S. Koladiya, R. Balasubramonian, C. Wilkerson, S. H. Pugsley, and Z. Chishti, "Efficiently prefetching complex address patterns," in *Proceedings of the 48th International Symposium on Microarchitecture*, ser. MICRO '15. New York, NY, USA: Association for Computing Machinery, 2015, pp. 141–152. [Online]. Available: https://doi.org/10.1145/2830772.2830793

[61] D. Joseph and D. Grunwald, "Prefetching using markov predictors," in *Proceedings of the 24th International Symposium on Computer Architecture*, ser. ISCA '97. New York, NY, USA: ACM, 1997, pp. 252–263. [Online]. Available: https://doi.org/10.1145/264107.264207

[62] L. Peled, U. Weiser, and Y. Etsion, "A neural network prefetcher for arbitrary memory access patterns," *ACM Transactions on Architecture and Code Optimization*, vol. 16, no. 4, pp. 37:1–37:27, 2019. [Online]. Available: https://doi.org/10.1145/3345000

[63] S. Pakalapati and B. Panda, "Bouquet of instruction pointers: Instruction pointer classifier-based spatial hardware prefetching," in *Proceedings of the 2020 47th International Symposium on Computer Architecture*, ser. ISCA '20. IEEE, 2020, pp. 118–131. [Online]. Available: https://doi.org/10.1109/ISCA45697.2020.00021

[64] A. V. Jamet, G. Vavouliotis, D. A. Jiménez, L. Alvarez, and M. Casas, "A two level neural approach combining off-chip prediction with adaptive prefetch filtering," in *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. Los Alamitos, CA, USA: IEEE Computer Society, March 2024, pp. 528–542. [Online]. Available: https://doi.org/10.1109/HPCA57654.2024.00046

[65] E. Bhatia, G. Chacon, S. Pugsley, E. Teran, P. V. Gratz, and D. A. Jiménez, "Perceptron-based prefetch filtering," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19. New York, NY, USA: Association for Computing Machinery, 2019, pp. 1–13. [Online]. Available: https://doi.org/10.1145/3307650.3322207

[66] A. Jain and C. Lin, "Linearizing irregular memory accesses for improved correlated prefetching," in *Proceedings of the 46th International Symposium on Microarchitecture*, ser. MICRO '13. New York, NY, USA: Association for Computing Machinery, 2013, pp. 247–259. [Online]. Available: https://doi.org/10.1145/2540708.2540730

[67] H. Wu, K. Nathella, D. Sunwoo, A. Jain, and C. Lin, "Efficient metadata management for irregular data prefetching," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19. New York, NY, USA: Association for Computing Machinery, 2019, pp. 1–13. [Online]. Available: https://doi.org/10.1145/3307650.3322225

[68] D. Kadjo, J. Kim, P. Sharma, R. Panda, P. Gratz, and D. Jimenez, "B-Fetch: Branch prediction directed prefetching for chip-multiprocessors," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '14, 2014, pp. 623–634. [Online]. Available: https://doi.org/10.1109/MICRO.2014.29

[69] L. M. AlBarakat, P. V. Gratz, and D. A. Jiménez, "SB-Fetch: synchronization aware hardware prefetching for chip multiprocessors," in *Proceedings of the 34th ACM International Conference on Supercomputing*, ser. ICS '20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 15:1–15:12. [Online]. Available: https://doi.org/10.1145/3392717.3392735

[70] A. Khadem, K. Kamalakkannan, Z. Zhu, A. Poptani, Y. Gu, J. B. Dominguez-Trujillo, N. Talati, D. Fujiki, S. Mahlke, G. Shipman, and R. Das, "DX100: Programmable data access accelerator for indirection," in *Proceedings of the 52nd Annual International Symposium on Computer Architecture*, ser. ISCA '25. New York, NY, USA: Association for Computing Machinery, 2025, pp. 1641–1658. [Online]. Available: https://doi.org/10.1145/3695053.3731015

[71] A. Saulsbury, F. Dahlgren, and P. Stenström, "Recency-based TLB preloading," in *Proceedings of the 27th International Symposium on Computer Architecture*, ser. ISCA '00. New York, NY, USA: Association for Computing Machinery, 2000, pp. 117–127. [Online]. Available: https://doi.org/10.1145/339647.339666

[72] G. B. Kandiraju and A. Sivasubramaniam, "Going the Distance for TLB Prefetching: An Application-driven Study," in *Proceedings of the 29th International Symposium on Computer Architecture*, ser. ISCA '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 195–206. [Online]. Available: http://dl.acm.org/citation.cfm?id=545215.545237

[73] A. Bhattacharjee and M. Martonosi, "Inter-core cooperative TLB for chip multiprocessors," in *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '10. New York, NY, USA: Association for Computing Machinery, 2010, pp. 359–370. [Online]. Available: https://doi.org/10.1145/1736020.1736060

[74] D. Chasapis, G. Vavouliotis, D. A. Jiménez, and M. Casas, "Instruction-aware cooperative TLB and cache replacement policies," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ser. ASPLOS '25. New York, NY, USA: Association for Computing Machinery, 2025, pp. 619–636. [Online]. Available: https://doi.org/10.1145/3669940.3707247

[75] A. V. Jamet, G. Vavouliotis, M. Torrents, D. Chasapis, and M. Casas, "Enhancing instruction prefetching via cache and TLB management," in *2026 ACM/IEEE 53rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2026, pp. 2254–2269. [Online]. Available: https://doi.org/10.1109/ISCA66397.2026.00159