ICP: Exploiting Instruction Correlation for Prefetching Irregular Memory Accesses

Mengming Li[1], Chenlu Miao[2], Buqing Xu[1], Qijun Zhang[1], Xiangfeng Sun[1], Ceyu Xu[1], Yuan Xie[1]
Wenkai Li[1], Shang Liu[1] and Zhiyao Xie[1*]

[1]The Hong Kong University of Science and Technology, [2]Independent Researcher
mengming.li@connect.ust.hk, chenlu.miao@outlook.com, {bxuax, qzhangcs, xsunbv}@connect.ust.hk,
{eeentropy, yuanxie}@ust.hk, {wlidm, sliudx}@connect.ust.hk, eezhiyao@ust.hk

***Abstract*—Irregular memory accesses pose challenges for effective and efficient data prefetching. While temporal prefetchers have recently shown promise for irregular memory access patterns, their effectiveness fundamentally depends on temporal address recurrence and large metadata storage. When memory addresses exhibit weak or no recurrence, as in indirect memory accesses, temporal prefetchers achieve limited performance gains while incurring substantial storage overhead.**

This paper proposes Instruction-Correlation Prefetching (ICP), a new hardware prefetching mechanism that exploits instruction-level correlations rather than memory-address correlations to handle irregular memory accesses. ICP observes that although memory addresses may not repeat, the instructions generating them often recur with stable data-dependency relationships. By learning these persistent instruction correlations, ICP speculatively computes and prefetches future irregular accesses using the execution results of their correlated predecessors. Across irregular SPEC CPU and GAP benchmarks, ICP outperforms the state-of-the-art temporal prefetcher Triangel by 14.0% and the indirect prefetcher DMP by 6.0%, while requiring only 2.1 KB of hardware storage, over three orders of magnitude smaller than temporal prefetchers.

I. Introduction

Cache prefetching, a long-standing technique for mitigating the "memory wall" problem [56], has been extensively explored as a means to improve processor performance. The fundamental challenge lies in efficiently predicting the diverse and complex memory access patterns that occur across different applications. To address this challenge, researchers have proposed various prefetching techniques tailored to distinct classes of access patterns [36], [21], [8], [59], [43], [34]. These prefetchers include stream prefetchers [30], [23], [25], stride prefetchers [9], [18], [33], spatial prefetchers [48], [32], [37], [41], [11], [13], [44], [47], [26], [29], indirect prefetchers [60], [28], [5], [31], [15], [17], [22], and temporal prefetchers [27], [42], [55], [51], [54], [10].

Among various prefetchers, **temporal prefetchers** represent a promising technique for addressing *irregular memory accesses*, where conventional prefetchers (e.g., stream, stride, spatial prefetchers) fall short. The core idea behind temporal prefetchers is to record previously accessed memory addresses and their correlated successor addresses as *metadata*. When a recorded address reappears during execution, prefetchers use this metadata to prefetch its correlated successor addresses.

*Corresponding Author

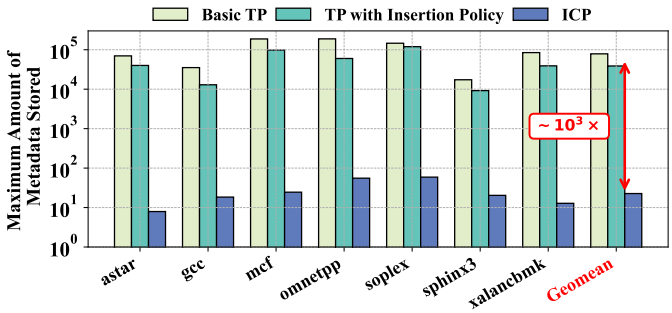

Fig. 1. Comparison of the maximum amount of metadata stored by ICP and temporal prefetchers (TP). ICP reduces the metadata overhead by three orders of magnitude.

Practically, the effectiveness of temporal prefetchers depends on two factors: 1) whether memory addresses involved in irregular access patterns exhibit temporal recurrence; 2) if such recurrence exists, how many of those addresses can be retained in table until their next appearance. These two factors fundamentally constrain the capability of temporal prefetchers.

Factor 1. Not all irregular memory accesses exhibit temporal recurrence. Currently, two classes of irregular memory accesses are commonly found in programs: *pointer accesses* and *indirect memory accesses*. Our evaluation shows that while temporal prefetchers are generally effective for pointer-based accesses, they fail to capture most indirect memory accesses[1]. Specifically, we evaluate a state-of-the-art temporal prefetcher on irregular sparse workloads dominated by indirect memory accesses [19], [20]. Our results show that it achieves only a 5.9% performance improvement over the baseline, whereas an indirect prefetcher yields a substantially higher 42.9% improvement. Furthermore, we observe that the metadata stored by temporal prefetchers exhibit a low reuse ratio (10^5 less than our solution) when processing indirect memory accesses. These observations all highlight their inherent limitation when memory addresses show weak or no temporal recurrence.

Factor 2. Temporal prefetchers suffer from significant storage overhead. Although temporal prefetchers effectively handle certain irregular memory access patterns (e.g., pointer-based accesses), they suffer from substantial metadata storage requirements, which are inherently difficult to reduce. This limitation arises because temporal prefetchers must record a large number of memory address correlations to maintain

[1]Detailed experimental results are presented in Section V.

sufficient coverage of recurring access patterns. To illustrate this challenge, we evaluate the maximum metadata storage required by temporal prefetchers on irregular SPEC CPU benchmarks. Two representative classes of temporal prefetchers are considered: *basic temporal prefetchers* (e.g., Triage [54]) and *enhanced temporal prefetchers* that incorporate non-recurrence metadata filtering strategies (e.g., Triangel [7]). As shown in Figure 1, both designs require metadata storage of the same order of magnitude, demonstrating that the large storage footprint is a fundamental limitation of temporal prefetching rather than an artifact of a specific design.

In response to the fundamental limitations of temporal prefetchers, we propose **ICP**, a new prefetching technique that exploits general instruction-level correlations rather than memory address correlations to handle irregular memory accesses. The core insight of ICP is that, although many irregular memory accesses lack temporal recurrence at the address level, the instructions generating these accesses often recur with consistent patterns during program execution. For example, consider two dependent load instructions, $PC_{\text{i}}: ld\ a1, 0(a0) \rightarrow PC_{\text{j}}: ld\ a2, 0(a1)$. Here, PC_{i} issues irregular memory accesses. While the specific memory addresses accessed by PC_{i} and PC_{j} may never repeat, their data-dependency relationship $a1$ remains stable across dynamic instances. By learning such persistent instruction correlations, ICP can speculatively compute and prefetch the memory addresses of irregular instructions using the execution results of their correlated predecessors.

ICP not only handles irregular memory accesses where temporal prefetchers fall short but also significantly reduces metadata storage requirements. As shown in Figure 1, ICP's metadata footprint is over three orders of magnitude smaller, i.e., $< 0.1\%$ of that required by temporal prefetchers. This efficiency stems from the fact that a single memory-access instruction may generate a vast number of dynamic requests. Consequently, one instruction-level correlation in ICP implicitly captures thousands or even millions of memory address correlations. By storing correlations at the instruction level rather than the address level, ICP achieves dramatically lower storage demands while maintaining comprehensive coverage of memory access patterns.

ICP is a *hardware-based* prefetching mechanism for irregular memory accesses. It consists of two primary components: PC Correlation Detection and Prefetching with PC Correlations. The PC Correlation Detection stage performs two key tasks: (1) it identifies memory instructions that are likely to generate irregular memory accesses (denoted as PC_{suc}) and their corresponding predecessor instructions that produce the required input values (denoted as PC_{pre}); and (2) it constructs the data-dependency paths starting from each PC_{pre} and ending at its dependent PC_{suc}: (PC_{pre}, PC_{suc}), and records them in the table. The Prefetching with PC Correlations stage then leverages cache line responses associated with PC_{pre} to speculatively compute memory addresses and prefetch data for the corresponding PC_{suc}. To enhance both prefetch coverage and timeliness, ICP simultaneously exploits cache lines fetched by demand and prefetch requests to drive the process.

Improvement over other prefetching solutions exploiting speculative instruction execution. ICP improves upon prior works by providing stronger generality for irregular memory accesses while maintaining moderate hardware complexity. This improvement stems from two key aspects of its design: (i) how it defines the speculated instruction chains and (ii) how it performs speculative execution to generate prefetches.

Definition of speculated instruction chains. State-of-the-art *indirect prefetchers* (e.g., Tyche [57] and DMP [19]) and *runahead execution* (e.g., VR [39] and DVR [40]) typically initiate chain discovery at a *striding load* and define the speculated chain as the set of dependent loads reachable from that striding load. This design choice is convenient, but also restrictive: being dependent on a striding load is neither a necessary condition nor a sufficient condition for a load to be an irregular. In contrast, ICP directly treats irregular memory instructions as prefetching targets and does not constrain the starting point of dependency discovery to striding accesses.

Mechanisms for speculative execution. *Indirect prefetchers* typically rely on the ability of a stride prefetcher to predict the full future address of striding loads. This allows them to obtain the data accessed by the striding load from prefetched cache lines and use it to speculatively execute the chain. As a result, their speculative execution model is tightly coupled to stride-driven instruction chains. *Runahead executions*, on the other hand, speculatively execute the dependency chains by leveraging spare core resources or another subthread. However, this capability comes at the cost of substantial hardware complexity, as it requires intrusive support in the CPU core to manage speculative execution. In contrast, ICP exploits cache-line responses associated with any existing prefetcher or demand request to trigger speculative execution, improving generality without incurring runahead-level complexity.

ICP outperforms the state-of-the-art temporal prefetcher Triangel [7] (14.0% performance speedup) and indirect prefetcher DMP [19] (6.0% performance speedup) across both irregular SPEC CPU benchmarks [2], which represent recurring irregular memory access patterns and have been evaluated in previous temporal prefetching works [7], [55], [54], [53], and GAP benchmarks [12], which primarily feature non-recurring indirect memory accesses and are widely used in indirect prefetching studies [19], [20], [28], [50]. ICP achieves these performance gains with only 449 B metadata storage and 1.7 KB control state, significantly lower than temporal prefetchers, which require up to 1 MB of metadata storage and 17.6 KB control state. Furthermore, our evaluations demonstrate that ICP achieves a substantially higher correlation reuse ratio than temporal prefetchers and exhibits greater correlation generality than indirect prefetchers.

II. Background and Motivation

A. Temporal Prefetching

Temporal prefetching has been proposed to address irregular memory access patterns that exhibit recurrence across time. The core idea of temporal prefetching [27], [42], [55], [51], [54], [10], [7], [53] is to record previously accessed memory

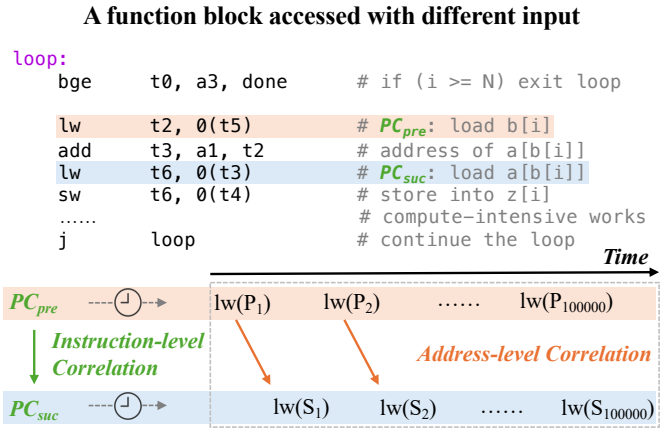

Fig. 2. Comparison between instruction-level correlations and memory-address-level correlations. Instruction-level correlations efficiently capture the patterns, whereas address-level correlations fail to do so.

addresses as metadata. When these addresses are accessed again, the prefetcher leverages the recorded metadata to predict subsequent memory accesses.

Existing temporal prefetchers face fundamental challenges:

Reliance on memory address repetition. Temporal prefetchers operate at the memory address level, which makes them ineffective at capturing correlations between memory access instructions when their accessed addresses do not recur over time. Figure 2 illustrates this problem. In the example, two dependent memory access instructions within a loop are repeatedly executed. However, since the loop itself is executed with different inputs, all memory addresses generated by these instructions appear only once and never recur in the future. As a result, applying temporal prefetchers to this pattern would fail to prefetch any memory access. Nevertheless, *dependence pattern between these two instructions does repeat* across loop iterations, yet temporal prefetchers fail to capture it.

Substantial metadata storage. All temporal prefetchers rely on large metadata structures to record correlations between memory addresses. Due to this significant capacity requirement, early designs placed metadata storage in off-chip DRAM. More recently, state-of-the-art prefetchers such as Triage [54], [53] and Triangel [7] co-locate metadata storage with the on-chip Last-Level Cache (LLC), thereby eliminating the need to fetch metadata from off-chip memory. However, this approach comes at a high cost: according to Triage and Triangel, the metadata storage can occupy up to half of the LLC capacity (i.e., 1 MB). To mitigate this overhead, prior work [36], [35] has proposed techniques for managing metadata storage more efficiently, including improved insertion and replacement policies as well as dynamic resizing. Nevertheless, even with these optimizations, metadata storage still requires hundreds of KB. This substantial requirement not only consumes valuable on-chip resources but also increases hardware complexity (e.g., enforcing way partitioning between the normal LLC and metadata storage).

B. Indirect Prefetching

Indirect prefetching [19], [50], [60], [28], [58], [4], [6] has been proposed to address nested array accesses, such as $a[b[i]]$. Indirect prefetchers typically operate in two stages: identification and prefetching. In the identification stage, they detect the innermost array access (e.g., $b[i]$) and its dependent outer array accesses (e.g., $a[b[i]]$). Since such nested array patterns are usually enclosed within loops, the innermost array access often exhibits regular stride behavior, which can be effectively captured by a stride prefetcher. Indirect prefetchers then monitor the data produced by the striding load and correlate these values with the addresses accessed by the dependent load to identify the outer array access. In the prefetching stage, indirect prefetchers leverage the data prefetched by the stride prefetcher ($b[i+d]$) together with the base address of the outer array to generate prefetch requests.

The primary limitation of indirect prefetchers lies in their ineffectiveness in handling memory access patterns beyond nested array accesses, as both their identification and prefetching mechanisms are tightly coupled to array semantics. In particular, indirect prefetchers assume that PC_{inner} (the load accessing the innermost array) corresponds to a striding load—even in Tyche [57], which aims to capture more general indirect patterns. This assumption provides several conveniences: (1) the indirection relationship can be identified by first detecting a striding load as PC_{inner} and then correlating the data it produces with addresses accessed by dependent loads; and (2) a stride prefetcher can predict the full memory address of PC_{inner}, allowing the line offset to be used to accurately extract accessed value from the prefetched cache line. However, once PC_{inner} does not exhibit regular stride accesses (e.g., *if (condition) x = a[b[i]].*), indirect prefetchers fail to identify and prefetch for such dependency patterns.

C. Runahead-based Schemes

Runahead-based schemes accelerate irregular workloads by speculatively pre-executing future instructions to compute and issue memory accesses ahead of demand. Vector Runahead (VR) [39] extends classic runahead by speculatively vectorizing future loop iterations into SIMD-style operations (e.g., gathers) so it can issue a batch of dependent misses in parallel and expose high MLP. Decoupled Vector Runahead (DVR) improves over VR by removing the key trigger limitation of conventional runahead: being constrained by the ROB-stall window. DVR decouples runahead from the main OoO pipeline and runs it in a subthread context, enabling more proactive lookahead and higher sustained MLP.

Runahead-based schemes incur significant hardware complexity. Both VR and DVR require substantial CPU-side modifications to support speculative execution and vectorization. DVR further introduces a decoupled subthread context to execute runahead slices, making them far more invasive than cache-local prefetchers. Moreover, VR and DVR initiate dependency-chain discovery from striding loads like indirect prefetchers, which inherently restricts their generality.

D. Motivations

Motivated by the fundamental limitations of existing temporal and indirect prefetchers, we introduce a new prefetching scheme for irregular memory accesses. The scheme works at the instruction level and is designed to capture general correlation patterns among *irregular* but *recurring* memory access instructions. We define two memory access instructions as **correlated** if a data-dependency path exists from one instruction to the other. Specifically, at runtime, we identify correlated and recurring memory instruction pairs $(PC_{\text{pre}}, PC_{\text{suc}})$, where PC_{suc} corresponds to irregular memory instructions, and PC_{pre} can be any memory instructions that producing data for PC_{suc}. The identified correlations are then maintained in a lightweight table. When PC_{pre} is re-executed after its correlations have been established, ICP leverages the execution results of PC_{pre} to speculatively execute the data-dependency path toward PC_{suc} and prefetch its corresponding memory accesses.

Advantages over temporal prefetching. Compared to temporal prefetching, our scheme works at the instruction level rather than the memory address level, offering the following advantages: (1) our scheme can capture certain patterns where temporal prefetching fall shorts. As shown in Figure 2, even if the memory addresses accessed by two correlated instructions do not recur in the future, the correlation between these instructions still repeat. Our scheme leverages this instruction-level recurrence to issue prefetches in such cases. (2) Our scheme requires significantly less storage than temporal prefetchers, as it only maintains instruction-level correlation information in tables without tracking the individual memory addresses they generate.

Advantages over indirect prefetchers. Our scheme can handle a much broader range of memory access patterns than indirect prefetching. We support any types of $(PC_{\text{pre}}, PC_{\text{suc}})$ pairs without restricting PC_{pre} and PC_{suc} to array access instructions. For example, PC_{pre} and PC_{suc} may correspond to pointer-based accesses, such as $*p$ and $(*p).next$. Alternatively, PC_{pre} and PC_{suc} may correspond to different types of accesses, such as an array-based access $a[i]$ and a pointer-based access $*a[i]$. In contrast, indirect prefetchers can only identify and prefetch a limited set of array-based pairs.

Advantages over runahead-based scheme. Compared to runahead-based approaches such as VR and DVR, ICP achieves substantially lower hardware complexity. Runahead schemes require non-trivial modifications to the CPU core to support speculative execution, vectorization, checkpointing, and recovery mechanisms. In contrast, ICP operates entirely within the cache hierarchy and interacts with the core in a non-intrusive manner, without modifying the CPU main pipeline. As a result, ICP provides better benefits for irregular memory accesses (Section V-B) while maintaining a significantly simpler, more modular, and easier-to-integrate hardware design.

III. Overview

In response to the motivations, we propose **ICP**, a new hardware prefetching scheme that records observed memory instruction correlations in a lightweight table and leverages this information to prefetch irregular memory accesses. Figure 3 illustrates the primary structures and workflow of ICP. The design includes two stages: *PC correlation detection* and *prefetching with PC correlations*.

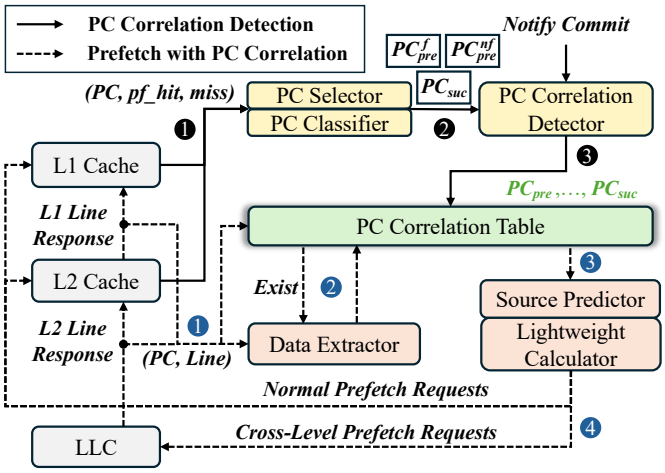

Fig. 3. ICP Overview.

A. PC Correlation Detection

In this stage, we identify correlated memory-instruction pairs $(PC_{\text{pre}}, PC_{\text{suc}})$, where PC_{suc} issues irregular memory accesses. To facilitate this process, we introduce the following hardware components:

PC Selector. This component is used to select PCs that are likely candidates for either PC_{pre} or PC_{suc} (Step ①).

PC Classifier. This component further classifies all PC_{pre} instructions into two categories: basic-prefetcher-friendly and non-basic-prefetcher-friendly. The rationale behind this classification is that the memory data of PC_{pre} can be obtained either from prefetch requests issued by basic hardware prefetchers or from demand accesses. Although both types of data can be used to execute the dependency path toward PC_{suc}, they differ in their impact on prefetching timeliness and accuracy. Data obtained from prefetch requests can initiate the execution of the dependency path earlier than demand accesses, thereby improving prefetching timeliness for PC_{suc}. However, inaccurate data generated by basic hardware prefetchers may ultimately lead to incorrect prefetches for PC_{suc}. To balance timeliness and accuracy, ICP adopts two distinct prefetching schemes for these categories: (1) if PC_{pre} can be effectively handled by basic prefetchers, its prefetched data is primarily used to compute the memory address of PC_{suc}, improving prefetching timeliness; and (2) if PC_{pre} cannot be effectively handled by basic prefetchers, ICP relies on the data from PC_{pre}'s demand accesses to prefetch for PC_{suc}, reducing inaccurate prefetches.

PC Correlation Detector. This component leverages the candidate sets of PC_{pre} and PC_{suc} provided by PC Selector and PC Classifier (Step ②) to identify correlated pairs $(PC_{\text{pre}}, PC_{\text{suc}})$. To establish correlations between PC_{pre} and its potential PC_{suc}, the PC Correlation Detector monitors committed instructions and employs a simplified $O(N)$ register renaming mechanism to construct data-dependency trees

rooted at PC_{pre} that extend to other PC_{suc} instructions. Each path from PC_{pre} to PC_{suc} defines a correlated instruction pair ($PC_{\text{pre}}, PC_{\text{suc}}$), consisting of a ***PC*pre node**, a ***PC*suc node**, and **intermediate instruction nodes** that form the dependency chain between them.

PC Correlation Table. This component records all identified correlated instruction pairs (Step ③) and serves as the interface between the *PC correlation detection* module and the *prefetching with PC correlation* module. Compared to the metadata table used in temporal prefetching, the PC Correlation Table introduces significantly lower storage overhead (less than 1 KB), as it only needs to record instruction-level correlations rather than memory address correlations.

B. Prefetching with PC Correlations

ICP monitors cache line responses (Step ①, dotted in blue). When PC_{pre} is re-executed after its correlations have been identified, the correlations stored in the PC Correlation Table are used to prefetch the upcoming memory accesses of PC_{suc}. To facilitate the prefetching process, we develop three hardware components: the Data Extractor, the Lightweight Calculator, and the Source Predictor.

Data Extractor. At Step ②, if the data response corresponds to a PC_{pre} entry recorded in the table, ICP employs Data Extractor to retrieve data accessed by PC_{pre} from the returned cache line. If PC_{pre} is identified as basic-prefetcher-friendly, ICP uses Data Extractor to predict the data within the cache line fetched by both prefetcher and demand accesses. Otherwise, ICP extracts data *only* from cache lines fetched by demand accesses.

Lightweight Calculator and Source Predictor. At Step ③, the Lightweight Calculator uses the data retrieved by Data Extractor to speculatively execute the instruction dependent on PC_{pre}. The dependent instruction may be an intermediate instruction along the dependency path or the PC_{suc} instruction itself. The Lightweight Calculator recursively executes the chained dependent instructions until the speculative execution reaches PC_{suc}. When source registers fall outside the identified dependency path, the Source Predictor predicts their corresponding values. Finally, at Step ④, ICP computes the target memory address of PC_{suc} and generates the corresponding prefetch request. When the data of PC_{pre} is obtained from a demand-fetched line, the prefetch request is forwarded to the LLC to enhance prefetch timeliness. Otherwise, ICP issues the prefetch request to the current cache level.

IV. Hardware Design

A. PC Selector and Classifier

The PC Selector and PC Classifier record performance counters of memory instructions to construct the $PC^{\text{f}}_{\text{pre}}$ (PC_{pre} and basic-prefetcher-friendly) candidate set, $PC^{\text{nf}}_{\text{pre}}$ (PC_{pre} and non-basic-prefetcher-friendly) candidate set, and the PC_{suc} candidate set. For each demand request arriving at ICP, three types of information are extracted: its PC address, whether it results in a prefetch hit, and whether it results in a demand miss. As shown in Figure 4, ICP maintains a Sample Table to

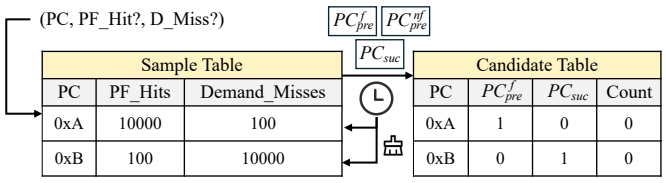

Fig. 4. PC Selector and Classifier.

track the performance counters. The Sample Table is indexed by the PC address of memory access instructions. When a PC experiences a prefetch hit or a demand miss, ICP increments the corresponding counter value in the Sample Table. To retain frequently accessed PCs, the Sample Table is managed using an LRU replacement policy. Due to distinct memory access patterns at different cache levels, ICP maintains separate instances of the Sample Table for the L1 and L2 caches.

The Sample Table periodically exports the PC_{pre} and PC_{suc} candidates for subsequent uses. ICP defines an epoch e as the number of times the Sample Table has been accessed. At the end of each epoch, ICP ranks all PCs based on their demand miss counts and evaluates the prefetching coverage of each PC. Specifically, ICP determines whether a PC qualifies as a PC_{suc} candidate using Equation 1:

$$\mathcal{S}_{suc} = \text{Top}_n\big(\{\, PC_i \mid \text{misses}(PC_i) \geq \theta_{\text{miss}} \,\}\big) \tag{1}$$

The formula selects n PCs with the highest demand miss counts as PC_{suc} candidates, where n is a design parameter controlled by the system designer. The rationale behind Equation 1 is that, in a system already equipped with basic prefetchers, PCs that still incur the most cache misses are likely not well served by those prefetchers and therefore are more likely to correspond to irregular memory accesses. ICP adjusts both the epoch length and the Sample Table size to ensure that the collected PCs accumulate sufficient demand misses for accurate classification. Specifically, prolonging the epoch length allows the Sample Table to gather enough statistical information, while restricting the Sample Table size ensures that PCs with fewer demand misses are gradually evicted and replaced by more memory-intensive ones. Additionally, ICP uses condition $\text{misses}(PC_i) \geq \theta_{\text{miss}}$ to filter out PCs with insignificant misses.

Then, ICP uses Equation 2 to determine whether a PC is selected as a $PC^{\text{f}}_{\text{pre}}$:

$$\begin{cases} \text{Cov}(PC_i) = \dfrac{\text{PF\_Hits}(PC_i)}{\text{PF\_Hits}(PC_i) + \text{Demand\_Misses}(PC_i)}, \\ \mathcal{S}^{f}_{pre} = \text{Top}_n(\{\, PC_i \mid \text{Cov}(PC_i) \geq \theta_{\text{cov}} \,\}) \end{cases} \tag{2}$$

The formula selects n PCs with the highest prefetching coverage as $PC^{\text{f}}_{\text{pre}}$ candidates. They could serve the potential data producer for PC_{suc}. Similar to Equation 1, ICP applies a threshold parameter θ_{cov} to filter out PCs with low prefetching coverage. In our experiments, we set $\theta_{\text{cov}} = 0.1$.

ICP leverages an important observation to construct the candidate set $\mathcal{S}^{nf}_{pre}$: **a *PC*suc can also serve as a *PC*pre**. Consider the nested pointer access pattern $*(*(*p))$. There are

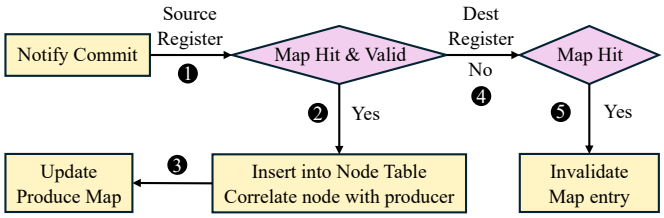

Fig. 5. Process of dependency tree construction.

three memory instructions, PC_1, PC_2, and PC_3, that access the pointers at p, $*p$, and $*(*p)$, respectively. PC_1 is regular because it originates from an array of pointer structures, and therefore serves as the PC_{pre} for PC_2. PC_2, which exhibits irregular behavior, acts as a PC_{suc} dependent on PC_1. Furthermore, PC_2 can also serve as PC_{pre} for the subsequent instruction PC_3. Driven by this insight, ICP directly reuses the set of $\mathcal{S}_{suc}$ as $\mathcal{S}^{nf}_{pre}$, as defined in Equation 3:

$$\mathcal{S}^{nf}_{pre} = \mathcal{S}_{suc} \tag{3}$$

Then, as shown in Figure 4, ICP employs another structure, Candidate Table (duplicated for L1 and L2 cache like Sample Table), to store identified $\mathcal{S}_{suc}$, $\mathcal{S}^{f}_{pre}$, and $\mathcal{S}^{nf}_{pre}$. Since the content of $\mathcal{S}_{suc}$ is identical to that of $\mathcal{S}^{nf}_{pre}$, ICP reduces the storage overhead by compressing Candidate Table and using a single field to indicate whether a PC entry belongs to $PC^{\text{nf}}_{\text{pre}}$ or PC_{suc}. The Candidate Table serves as the interface between PC Selector/Classifier and PC Correlation Detector. After storing these three sets into Candidate Table, ICP clears Sample Table and continues searching for potential PC_{pre} and PC_{suc} instructions in the next execution phase.

B. PC Correlation Detector

The PC Correlation Detector leverages the candidate sets stored in Candidate Table to identify correlated instruction pairs (PC_{pre}, PC_{suc}). To discover these correlations, ICP constructs data-dependency trees rooted at each PC_{pre}, which expands toward its dependent PC_{suc} instructions. Each path from a PC_{pre} node to a PC_{suc} node represents a correlated instruction pair (PC_{pre}, PC_{suc}).

1) Dependency Tree Construction Triggering: The PC Correlation Detector leverages the committed instruction information from the ROB to construct dependency trees. The construction process is triggered whenever the PC of a committed instruction belongs to either $PC^{\text{f}}_{\text{pre}}$ or $PC^{\text{nf}}_{\text{pre}}$. Once a dependency tree construction is initiated, ICP temporarily blocks subsequent construction requests, even if they correspond to another PC_{pre}. This blocking mechanism prevents resource contention. To avoid a scenario where a single PC_{pre} monopolizes hardware resources and causes deadlock, ICP introduces a `Count` field in the Candidate Table. This field records the number of attempts made to construct the dependency tree for the corresponding PC. When the count exceeds a predefined threshold, the PC Correlation Detector ceases further construction attempts for that PC.

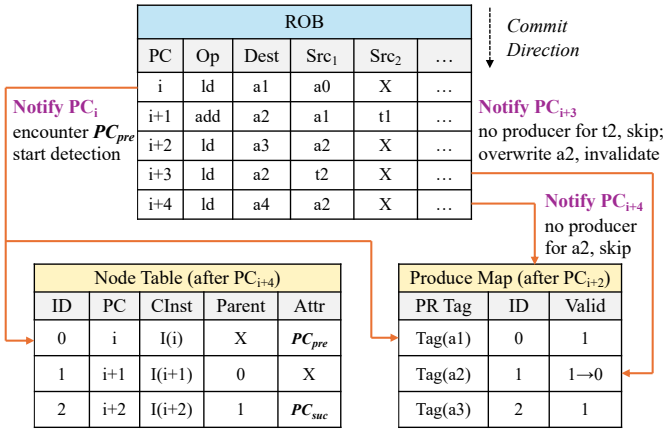

Fig. 6. Example of dependency tree construction.

2) Dependency Tree Construction: As shown in Figure 6, ICP employs two hardware structures: the Node Table and the Produce Map, to construct the dependency tree rooted at the triggered PC_{pre}. The **Node Table** primarily records correlations between different PCs within the dependency tree (through the `ID` and `Parent` fields) and stores auxiliary information that enables reconstruction of the instruction-level dependency path from PC_{pre} to PC_{suc} (through the `PC`, `CInst`, and `Attr` fields). By hitchhiking on the ROB commit process, ICP adopts the core concept of register renaming and implements a simplified $O(N)$ register renaming scheme to efficiently track data dependencies among committed instructions during dependency tree construction. The **Produce Map** maintains the mapping between physical registers and their producer PC nodes. Specifically, it tracks which PC node last produced the value for each physical register (`PR Tag`), represented by the `ID` field in the Node Table, and uses the `Valid` field to indicate whether that mapping is currently valid. We now describe the process of dependency tree construction as follows:

First, upon triggering dependency tree construction, ICP initializes the Node Table and Produce Map using the committed instruction at the head of the ROB. ICP allocates a new entry in the Node Table with ID = 0 and records its compressed instruction format (detailed in Section IV-C), PC address, and attribute, which indicates whether the PC belongs to $PC^{\text{f}}_{\text{pre}}$, $PC^{\text{nf}}_{\text{pre}}$, or PC_{suc}. Then, as shown in Figure 5 and Figure 6, ICP updates the Node Table and Produce Map using information from subsequently committed instructions.

1) **Checking Source Registers.** Each source register of the committed instruction notified from the ROB is sent to the Produce Map for lookup.
2) **Updating Node Table.** If any source register hits in the Produce Map, the committed instruction is determined to be data-dependent on the producer instruction indicated by the corresponding `ID` entry in the Produce Map (as illustrated by PC_{i+1} and PC_{i+2} in Figure 6). In such case, ICP inserts a new entry at the end of the Node Table and increments the node `ID` by one. It sets the `Parent` field of the newly inserted node to the node

ID of its producer instruction. If the instruction belongs to PC_{suc}, ICP updates its Attr field accordingly.

3) **Updating Produce Map.** ICP updates the Produce Map using the destination register of the committed instruction. The ID field in the Produce Map is set to the corresponding ID previously allocated in the Node Table, and the Valid field is initialized to 1.
4) **Checking Destination Register.** If no correlation with previous instructions is found for the current committed instruction, ICP does not maintain any information about this instruction in either the Node Table or the Produce Map (PC_{i+3} and PC_{i+4} in Figure 6).
5) **Invalidating Map Entry.** Although ICP skips committed instructions without correlations, such instructions may overwrite physical registers whose values were produced by earlier instructions already tracked in the Produce Map (e.g., PC_{i+3}). In these cases, ICP invalidates the corresponding entry in the Produce Map to maintain the correctness of data-dependence correlations.

ICP defines two termination conditions for the above process. First, ICP limits the maximum number of committed instructions processed during each dependency tree construction trigger. Our experiments set this limit to 128. Second, ICP restricts the size of Node Table. When the number of recorded nodes reaches the maximum capacity of Node Table, ICP terminates the dependency tree construction for current PC_{pre}. Our experiments set Node Table size to 16. The above two termination conditions are combined using a logical OR.

3) Correlated Instruction Pairs Reconstruction: ICP traverses the *Node Table* to identify and generate correlated instruction pairs (PC_{pre}, PC_{suc}). With a Node Table capacity of $N = 16$, reconstructing all dependency paths from the root (PC_{pre}, entry $\texttt{ID} = 0$) to every PC_{suc} requires:

1) a single linear scan of the table to collect the PC_{suc} entries ($\leq N - 1$); and
2) for each PC_{suc} entry, following its Parent pointers back to $\texttt{ID} = 0$.

The total computational effort corresponds to the aggregate of all path lengths from each PC_{suc} node to the root. In the worst-case scenario, this overhead can be expressed as:

$$\sum_{d=1}^{N-1} d = \frac{N(N-1)}{2}$$

When $N = 16$, we require at most 120 parent-pointer dereferences. Assuming a 1-cycle SRAM access per table lookup, the worst-case traversal requires no more than 120 cycles and utilizes only a small temporary buffer (up to 16 IDs) to record the path.

In practice, the dependency trees are typically shallow (Figure 13). By performing a one-time depth-first search (DFS) from the root node (PC_{pre}) to all reachable PC_{suc} nodes, ICP reduces the total number of table reads to $O(N)$ (i.e., at most 16 cycles for 16 table lookups) to reconstruct all paths. Consequently, the path reconstruction overhead is negligible relative to overall program execution time.

```
0xA: lw t2,0(t5) -> 0xB: add t3,a1,t2 -> 0xC: lw t6,0(t3)
```

Correlation Table						
PC Tag	Counter	Friendly?	Level	Corr PC	Corr Inst	Src Pred?
0xA	1	1	1	0xB	COP = add Imm = 0	1
	…	0	2	0xDE	…	…
0xB	1	0	1	0xC	COP = ld Imm = 0	0
	X	X	X	X	X	X

Fig. 7. Correlation Table structure and usage.

C. PC Correlation Recording

ICP employs the Correlation Table to record the instruction correlations identified in Section IV-B. Figure 7 illustrates the table structure and its usage. For each PC entry, ICP stores up to two successor instructions that consume the value produced by this PC. This choice reflects two realities: (i) a producer instruction may have multiple consumers, and (ii) program phase behavior can cause different successors to be observed across execution epochs. Consequently, retaining more than one successor per PC increases prefetching coverage while keeping storage overhead bounded. To handle cases where a PC has more than two successor instructions, ICP employs the Counter field in the Correlation Table to retain the most frequently observed successors. When a new successor is identified and all successor slots are occupied, the entry with the smallest Counter value is replaced. The Friendly field indicates whether the recorded PC is classified as basic-prefetcher-friendly. This information is later utilized during the prefetching phase for selecting cache line responses (from prefetches or demand accesses). The Level field associates each correlated instruction pair with its corresponding cache level. Since ICP maintains separate instances of the Sample Table and Candidate Table for different cache levels, this field is set to the cache level from which the correlated instruction pair is derived. During the prefetching phase, ICP uses the cache line response from the cache level indicated by the Level field to speculatively execute the successor. The Corr PC field stores the PC address of the successor instruction. The Corr Inst field records the essential information of the successor instruction, including the operation type supported by the Lightweight Calculator in ICP and its corresponding immediate value. ICP excludes any dependency paths containing operation types that are not supported by the Lightweight Calculator. Furthermore, a correlated instruction may contain multiple source registers, some of which may lie outside the identified dependency path. To enable speculative execution of such instructions during the prefetching phase, ICP employs the Source Predictor to estimate the values of these external source registers. The Src Pred field indicates whether the execution of a given correlated instruction requires this prediction mechanism.

D. Prefetching with PC Correlations

During the prefetching phase, ICP monitors cache-line responses originating from both prefetch and demand requests. If a response corresponds to a PC entry recorded in the

TABLE I
OPERATIONS SUPPORTED BY THE LIGHTWEIGHT CALCULATOR.

Category	Operation	Example
Arithmetic	ADD	$x = a + b$
	SUB	$x = a - b$
	SHL	$x = a \ll k$
	SHR	$x = a \gg k$
Logical	AND	$x = a \ \& \ b$
	OR	$x = a \mid b$
	XOR	$x = a \oplus b$

Correlation Table (excluding prefetched lines associated with non–basic-prefetcher-friendly PCs), ICP utilizes the data contained in that cache line to speculatively execute the successor instruction associated with the PC and ultimately issue prefetch requests for the PC_{suc}. To support this process, ICP integrates three specialized components: the **Data Extractor**, which retrieves the data accessed by the PC within the cache line; the **Lightweight Calculator**, which performs speculative execution of the correlated successor instructions; and the **Source Predictor**, which predicts the values of source registers when a successor instruction contains operands outside the identified data-dependency path.

Data Extractor applies different data extraction methods depending on whether the cache line is fetched by a demand request or a prefetch request. If the cache line is fetched by a demand request, Data Extractor directly utilizes offset bits of the memory access address to locate and extract the target data from the cache line. Conversely, if the cache line is fetched by a prefetch request, Data Extractor must infer the offset of the target data within the line, as most cache prefetchers operate at the cache-line granularity and prefetch requests record only the line address without fine-grained offset information. To handle such cases, Data Extractor maintains a table that records historical line offsets observed from demand requests, which are then used to infer the offset for prefetched cache lines. Specifically, when a basic-prefetcher-friendly PC is added to Correlation Table, ICP simultaneously allocates an entry in Data Extractor. The Data Extractor associates each observed line offset with a counter to track its occurrence frequency. During the prefetching phase, it computes the relative probability of each offset by dividing its individual counter value by the sum of all counters across offsets. When the probability of a line offset exceeds a predefined threshold (set to 0.1 in our experiments), Data Extractor uses the corresponding offset to extract the target data from the prefetched cache line.

The Data Extractor enables ICP to trigger speculation from cache-line responses that arrive *far ahead* of PC_{pre}, improving timeliness relative to indirect prefetchers. This is because indirect prefetchers typically rely on a stride prefetcher to predict the *exact* address and extract the accessed word. With the same prefetch degree, a line-level prefetcher can generally fetch cache lines farther ahead than an exact-address stride prefetcher. Data Extractor allows ICP to utilize cache-line responses generated by *any* existing prefetcher as valid triggers, enabling earlier and more general prefetches.

Lightweight Calculator leverages the data provided by the Data Extractor and the successor instruction information stored in the Correlation Table to speculatively execute the dependency path starting from PC_{pre}. When the successor instruction is a memory access instruction, the Lightweight Calculator computes its target memory address and issues the corresponding prefetch request. Otherwise, the Lightweight Calculator recursively continues speculative execution along the dependency path. It is designed to support basic arithmetic and logical operations (Table I), as intermediate instructions between two correlated memory instructions are typically simple address-calculation operations. Our extensive evaluation results (Section V) confirm that this operation set is sufficient to achieve high performance. Consequently, the Lightweight Calculator introduces limited hardware complexity.

Source Predictor is employed to predict source register values when speculatively executing successor instructions with the `Src Pred` flag set. Since the intermediate instructions between PC_{pre} and PC_{suc} are memory address computation operations, the external source registers involved typically hold stable base addresses. Leveraging this property, Source Predictor leverages historical source register values to perform value prediction. Specifically, when a PC marked as `Src Pred` is added to the Correlation Table, ICP simultaneously allocates a corresponding entry in Source Predictor. Each entry records the PC address and the target source register to be predicted. During execution, Source Predictor monitors ROB to track committed values of corresponding source registers. It maintains a single recorded value for each tracked source register, along with a confidence bit that indicates prediction stability. When a newly observed value matches the recorded one, the confidence bit is set; otherwise, it is reset. The Source Predictor produces a prediction only when the confidence bit is set, and the predicted value is then used for speculative execution of the successor instructions.

E. Integration with CPU and Memory Hierarchy

According to Figure 3, ICP interfaces with both the CPU core and the memory hierarchy and relies on three categories of information: (1) demand requests, (2) cache-line responses returned from the memory hierarchy, and (3) committed instruction information from the CPU core. Since receiving demand requests is already a standard capability of hardware prefetchers, we focus primarily on how ICP accepts cache-line responses and committed instructions.

Acquiring cache-line fill information, including both the returned data and the associated PC of the triggering memory instruction, is a common requirement in existing indirect prefetchers [19], [57], [58] and other cache-related microarchitecture designs [52], [49]. Basically, returned data can be obtained by snooping the data bus [19]. To propagate the associated PC, we extend each MSHR target entry with a compressed PC field, which is returned alongside the filled data. For compression, we observe that instructions within the same dependency chain often share identical high-order bits in their PC addresses. Therefore, we retain only the lower 3–4

bits of each PC and hash the high-order bits. Consequently, each compressed PC address requires only 10 bits. The storage overhead of this PC plumbing is: $N_{\text{MSHR}} \times T \times \text{PC}_{\text{bits}}$, where N_{MSHR} is the number of MSHRs and T is the number of targets per MSHR. For example, with $N_{\text{MSHR}} = 16$, $T = 8$, and a 10-bit PC, the additional storage is $16 \times 8 \times 10 = 1280$ bits, i.e., 160 B. This overhead is minimal.

Similarly, leveraging committed instruction information is not unprecedented in prefetcher designs [3], [21], [45]. Importantly, ICP imposes very loose timing requirements on commit information because (1) the entire dependency chain construction process operates outside the CPU's critical execution path (Section IV-B); (2) Once the dependency path between $(PC_{\text{pre}}, PC_{\text{suc}})$ is learned, it is recorded and reused for subsequent executions. Any delay in completing path construction merely postpones the issuance of prefetches for PC_{suc} by a limited number of cycles. This loose timing constraint allows a clean decoupled interface. A small FIFO buffer can be inserted between the commit logic and ICP to asynchronously stream committed instruction information. The commit logic enqueues commit records, while ICP dequeues them at its own pace to construct dependency trees.

V. Evaluation

A. Experimental Setup

TABLE II
System Configuration.

Module	Configuration
Core	5-wide fetch, 5-wide decode 10-wide issue, 10-wide commit 120-entry IQ, 85/90-entry LQ/SQ 288-entry ROB, L-TAGE branch predictor
Private L1 I/D cache	64 KB each, 4-way, 64B line, 16 MSHRs PLRU, 2 cycles hit latency Stream, stride, and spatial prefetchers scheduled by Alecto [36] for L1D cache
Private L2 cache	512 KB, 8-way, 64B line, 32 MSHRs PLRU, 9 cycles hit latency, inclusive
Shared L3 cache	2 MB/core, 16-way, 64B line, 36 MSHRs CHAR [16], mostly_exclusive 20 cycles hit latency
Memory	LPDDR5_5500_1x16_BG_BL32

System Configuration. We evaluate ICP using the full-system (FS) mode of gem5 [14], with baseline system parameters summarized in Table II. The configuration is designed to closely follow the original Triangel [7] implementation, with the remaining parameters chosen to approximate those of the Arm Cortex X2. To emulate a realistic prefetching environment similar to that of commercial CPUs, we integrate a hybrid configuration comprising multiple basic hardware prefetchers. We deploy stream, stride, and spatial prefetchers from IPCP [43] at the L1 cache, and coordinate them using Alecto [36], a state-of-the-art prefetcher selection algorithm. These baseline prefetchers effectively capture regular memory access patterns but remain ineffective for irregular accesses, which ICP and other evaluated prefetchers target.

Evaluated Prefetchers. We compare ICP against representative prefetchers targeting irregular memory access patterns, including the state-of-the-art *temporal prefetcher* Triangel [7], the *indirect prefetcher* Tyche [57] and DMP [19], the *runahead scheme* Vector Runahead [39] and Decoupled Vector Runahead [40], and the integrated design DMP+Triangel. For fair comparison, ICP and indirect prefetchers are both integrated at the L1 cache level. Triangel is integrated at the L2 cache level, following its original design [7]. Runahead-based schemes are integrated with the CPU main pipeline.

Workloads. We evaluate ICP using irregular SPEC CPU benchmarks [2], which are commonly used to assess existing temporal prefetchers [55], [54], [53], [7], and the GAP benchmark [12], which is widely adopted by indirect prefetchers [28], [5], [19], [50]. For GAP, we use the same input graphs as DMP [19]. We apply the SimPoint technique [46] to generate checkpoints across all workloads. Each SimPoint-sampled checkpoint is warmed up with 200M instructions, followed by a simulation of the next 20M instructions. The reported metrics for each workload are calculated by aggregating the results from all its checkpoints with weighted averages.

B. Performance Evaluation

Figure 8 presents the performance speedup of ICP compared to other prefetchers designed for irregular memory access patterns. The evaluation spans two benchmark suites: SPEC, which features diverse and complex memory access patterns, and GAP, which primarily exhibits non-recurrent memory addresses but stable indirect instruction-level dependencies. The results demonstrate that ICP demonstrates consistent effectiveness across both SPEC and GAP benchmarks. ICP achieves a 25.51% performance improvement over the baseline system equipped only with basic prefetchers, outperforming Triangel by 13.99%, Tyche by 10.86%, DMP by 5.97%, VR by 15.03%, DVR by 3.74%, and performing on par with the combined DMP+Triangel configuration (25.61%).

Temporal prefetchers perform well on SPEC benchmarks but achieve poor performance across most workloads in the GAP benchmark suite. In particular, Triangel struggles to capture irregular memory accesses that exhibit instruction-level repetition but lack explicit address-level repetition, which commonly occurs in GAP workloads (Section V-C).

Indirect prefetchers are ineffective for many workloads in the SPEC benchmarks. Specifically, they fail to identify and prefetch more complex irregular access patterns that extend beyond nested memory accesses in SPEC (Section V-D).

Runahead-based schemes deliver effective performance gains only when sufficient runahead time is available. Thus, VR fails to achieve strong performance across both SPEC and GAP. Although DVR improves over VR, it still struggles with workloads that exhibit complex dependency patterns, such as *mcf* and *gcc*. This limitation arises because DVR identifies dependency chains starting from striding loads, similar to indirect prefetchers. Moreover, a key drawback of runahead-based approaches is their substantial hardware complexity, as they require tight integration with the core pipeline (Section V-E).

Shortcomings of integrating indirect prefetchers with temporal prefetchers. Although integrating DMP with Triangel achieves a performance gain comparable to ICP, this

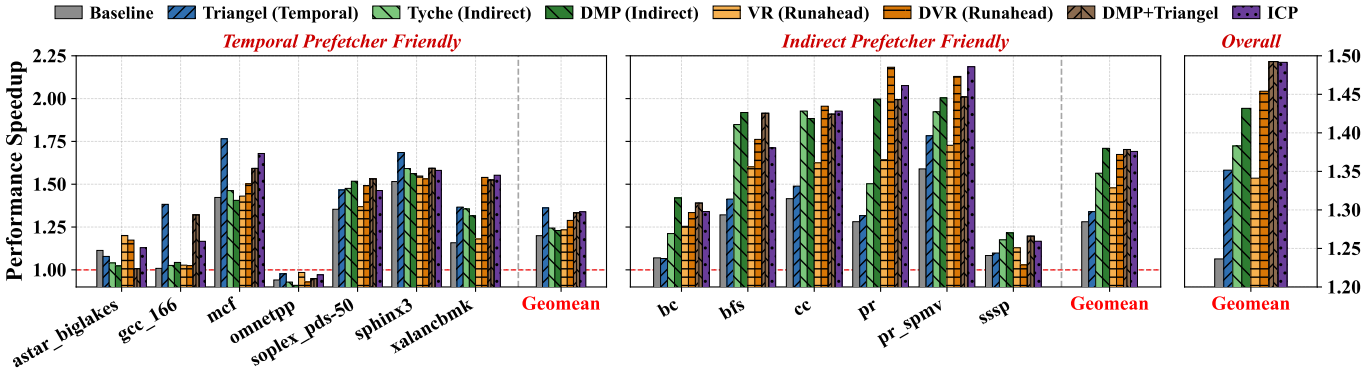

Fig. 8. IPC speedup on SPEC and GAP benchmarks. ICP outperforms Triangel by 13.99%, Tyche by 10.86%, DMP by 5.97%, VR by 15.03%, and DVR by 3.74%. While ICP attains performance comparable to the combined DMP+Triangel configuration, it provides several key advantages: 1) Substantially smaller storage overhead; 2) Lower hardware complexity; 3) Lower DRAM traffic; 4) Higher energy efficiency.

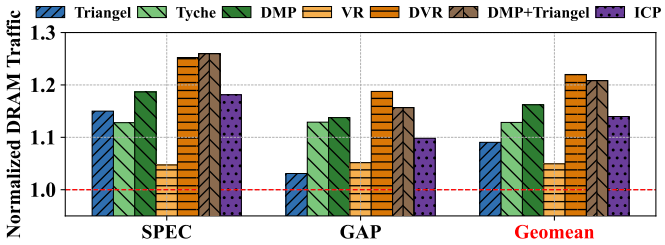

Fig. 9. Comparison of DRAM traffic. ICP introduces 6.84% less DRAM traffic than the DMP+Triangel hybrid configuration, 8.02% than the DVR.

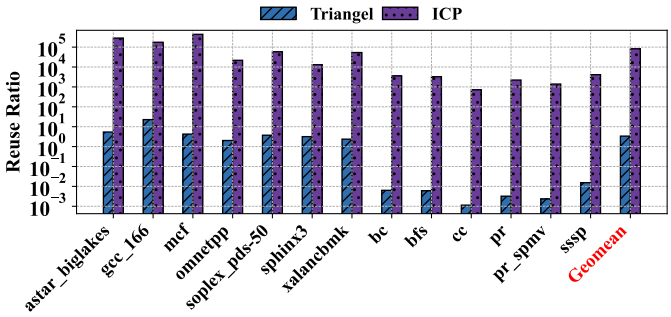

Fig. 10. Comparison of metadata reuse ratio between ICP and Triangel. The reuse ratio in ICP is on the order of 10^5 times higher than that of Triangel.

hybrid configuration inherits the limitations of both prefetching designs. **First**, the hybrid design incurs significantly greater hardware overhead than ICP. Specifically, Triangel and other temporal prefetchers typically require substantially larger metadata storage (up to 1MB), along with additional metadata management strategies (e.g., 17.6KB in Triangel [7]). DMP additionally requires 912B storage. In contrast, ICP only requires 1.6 KB storage in total (Section V-I). **Second**, the hybrid design introduces significant hardware complexity. For instance, to support temporal prefetchers, the system must share the LLC space with the metadata table, necessitating partitioning schemes and specific metadata table access, insertion, and replacement strategies. **Third**, the hybrid design causes higher DRAM traffic than ICP. Figure 9 illustrates the DRAM traffic for each evaluated prefetcher configuration, normalized to the baseline. The results indicate that ICP introduces a 13.98% increase in DRAM traffic over the baseline system, compared to 9.04% for Triangel, 16.21% for DMP, and 20.82% for the DMP+Triangel combination. Consequently, DMP+Triangel incurs 6.84% more DRAM traffic than ICP. **Fourth**, the hybrid design is less energy efficient, consuming 4.9% more enegy than ICP (Section V-J).

C. ICP vs Temporal Prefetchers: More Efficient Correlations

To demonstrate the efficiency of instruction-level correlations, Figure 10 compares the metadata reuse ratio between ICP and Triangel. In this comparison, the metadata in ICP corresponds to instruction-level correlations, whereas in Triangel it represents address-level correlations. The reuse ratio is computed as the total number of metadata accesses divided by the number of metadata insertions. A higher reuse ratio indicates that the stored metadata is accessed more frequently after being recorded, reflecting higher metadata utilization.

The results show that the reuse ratio of metadata in ICP is on the order of 10^5 times higher than that of Triangel. Combined with Figure 1, our evaluations demonstrate that instruction-level correlations are substantially more efficient than address-level correlations in capturing irregular memory access patterns. Furthermore, the metadata reuse ratio of Triangel in GAP benchmarks is extremely low, indicating that memory addresses in these workloads are rarely re-accessed. This observation aligns with the performance results shown in Figure 8, where Triangel delivers limited benefits on GAP. In contrast, instruction-level correlations in them frequently reoccur, which validates ICP's strong performance in GAP.

D. ICP vs Indirect Prefetchers: More General Correlations

To demonstrate the broader generality of ICP compared to indirect prefetchers, Figure 11 compares the total number of identified correlations across SPEC benchmarks, where Tyche and DMP exhibits limited effectiveness (Figure 8). The correlations identified by ICP represent general instruction-level relationships $(PC_{\text{pre}}, PC_{\text{suc}})$. In contrast, the correlations identified by Tyche correspond to *IMA* patterns, while those in DMP correspond to *indirection pairs* [19], both of which are specifically designed to capture data-dependence relationships between two nested array access instructions.

We find that DMP's differential matching mechanism [19] can occasionally capture data dependencies between memory instructions beyond strictly nested array accesses, such

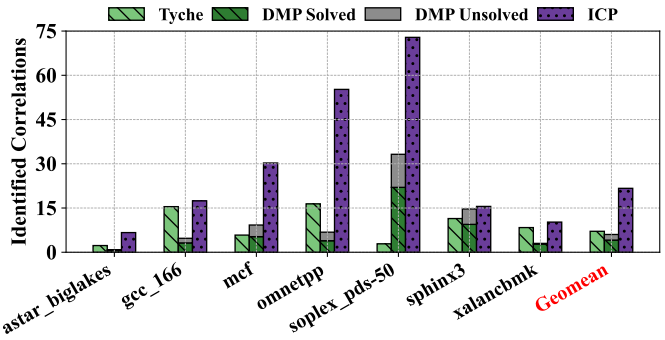

Fig. 11. Comparison of identified instruction correlations between ICP and indirect prefetcher. ICP can identify and handle more general correlations.

as data dependencies in array-of-pointers structures (e.g., $p[i] \rightarrow *p[i]$). However, not all such patterns can be effectively handled by DMP, since its prefetching mechanism is explicitly designed for nested array structures. To highlight this limitation, Figure 11 categorizes the indirection pairs identified by DMP into two groups: *solved* and *unsolved*, representing whether DMP can successfully prefetch for them.

Our results show that ICP identifies substantially more correlations than indirect prefetchers across the SPEC benchmarks, with an average of 22 correlations, compared to 7 for Tyche and 6 for DMP. Furthermore, nearly one-third (2 out 6) of the indirection pairs identified by DMP are ineffective. These findings underscore ICP's effectiveness in handling complex and diverse irregular memory access patterns.

E. ICP vs Runahead: Lower Hardware Complexity

Table III compares ICP and DVR in terms of integration with CPU core and memory hierarchy. Overall, ICP achieves substantially lower complexity, whereas DVR requires tight coupling with the CPU pipeline and thread support.

Thread requirement. ICP does not require any additional hardware thread contexts. In contrast, DVR relies on a dedicated subthread to execute the decoupled runahead/discovery stream, introducing extra control for thread management.

Decode/execute modifications. ICP incurs no changes to the core's decode and execute stages. DVR, however, requires a mode-aware decode path (Discovery vs. DVR execution) and execution support for DVR-generated vectorized operations.

Commit requirements. ICP only needs committed instruction information with loose timing. This enables a clean decoupling via an asynchronous buffer between the commit stage and ICP. DVR instead must ensure that DVR-generated operations do not update architectural state and must additionally provide termination/recovery logic when DVR stops.

Memory interface. ICP uses a standard cache prefetcher interface. DVR, however, introduces additional throttling mechanisms to detect memory pressure and regulate the potentially high bandwidth demand generated by vectorized runahead.

F. Analysis of ICP Characteristics

In this section, we examine two key characteristics of ICP: the **instruction correlation learning rate**, which reflects the time required to detect instruction correlations, and **the length of the derived dependency paths**, which represents the total number of instructions needed to compute the path from PC_{pre} to PC_{suc}. A higher learning rate allows ICP to identify instruction correlations and initiate prefetching more quickly. Conversely, a shorter dependency path length enables ICP to compute the address of PC_{suc} more efficiently.

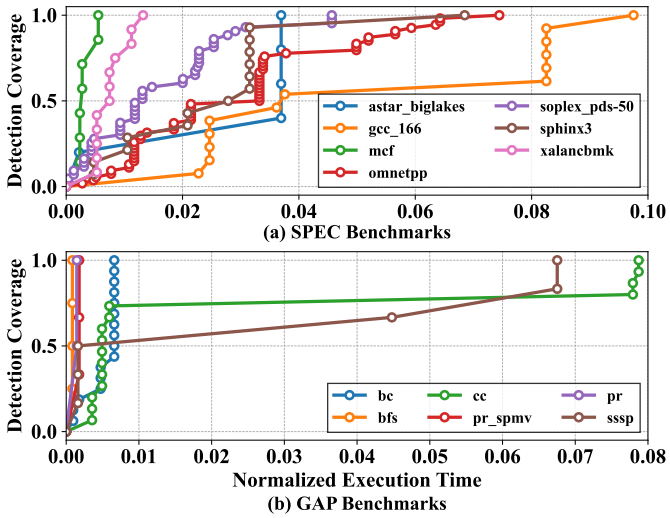

Fig. 12. The learning rate of ICP. ICP completes the identification of all instruction correlations within less than 10% of total program execution time.

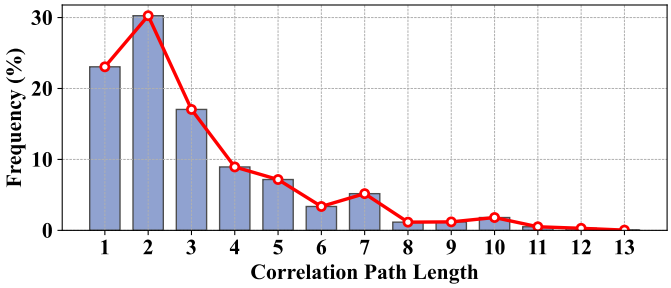

Fig. 13. The dependency path length distribution. ICP completes the majority of its prefetching process within only a few cycles.

1) Instruction Correlation Learning Rate: Figure 12 shows the detection coverage, defined as the ratio of covered correlations to the total number of identified correlations at the end of the programs, plotted against the program execution time normalized to the total execution time for ICP. The results demonstrate that ICP completes the identification of all instruction correlations within less than 10% of total program execution time across both SPEC and GAP. Moreover, most correlations in SPEC are detected within the first 4% of execution, while the majority in GAP are identified in under 1%. By comparing the SPEC and GAP results, we find that the learning time increases with the complexity of memory access patterns. This is because instruction correlations may vary across different execution phases. ICP can only observe and record a correlation when the corresponding execution phase is encountered for the first time.

2) Dependency Path Length: Figure 13 shows the distribution of dependency path lengths across the SPEC and GAP benchmarks. The results reveal that most dependency paths are short: over 70% of the identified paths have a length

TABLE III
INTEGRATION COMPLEXITY COMPARISON BETWEEN ICP AND RUNAHEAD SCHEMES.

Integration aspect	ICP (this work)	DVR
Thread requirement	Not required.	Required.
Decode	No changes.	Mode-aware (Discovery or DVR) decode path.
Execute	No changes.	Support for DVR-generated vectorized address-generation ops.
Commit	Asynchronous Buffer.	Ensure DVR-generated ops not update architectural state Termination/recovery logic When DVR stops
Memory interface	Demand training and cache fill interface.	Throttling mechanisms tied to memory pressure.

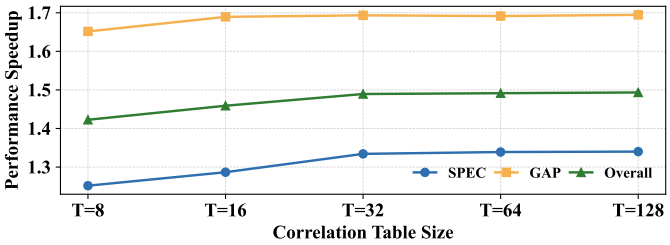

Fig. 14. Performance impact of Correlation Table size.

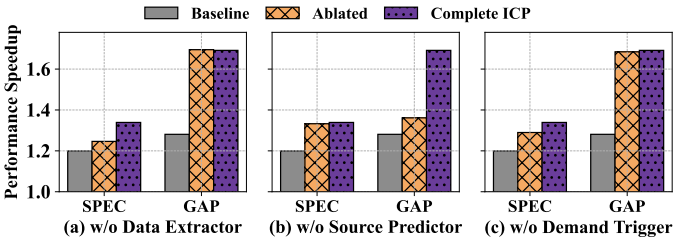

Fig. 15. Ablation study of ICP.

of no more than three instructions. This indicates that the majority of the speculation process initiated by ICP (i.e., the time from receiving cache responses from PC_{pre} to issuing prefetch requests for PC_{suc}) requires the execution of fewer than 3 instructions. Additionally, the longest dependency path identified by ICP consists of only 13 instructions. Since ICP handles only basic arithmetic and logical operations (Section IV-D) and excludes all others, the speculative execution process typically takes between 1 to 3 cycles, with a maximum of 13 cycles. Given that cache prefetchers operate off the core's critical execution path and typical memory access latencies span hundreds of cycles, this additional latency is negligible for prefetching timeliness and well within acceptable bounds.

3) Correlation Table Size: Figure 14 evaluates the performance impact of Correlation Table size. We vary the number of entries T from 8 to 128. The results show that performance improves as the table size increases from $T = 8$ to $T = 32$, as a larger table can store more instruction-level correlations discovered during execution. However, increasing the table size further to $T = 128$ yields only marginal improvement, demonstrating clear diminishing returns. The results highlight that the number of active instruction correlations within a program phase is relatively small, and a lighweight Correlation Table is sufficient to store them.

G. Ablation Study

Impact of Data Extractor. Figure 15(a) shows performance on SPEC degrades noticeably when Data Extractor is removed. Data Extractor extracts the accessed value from prefetched cache lines by predicting the offset. Without it, ICP must consider all possible offsets within the cache line to extract values. This behavior significantly increases the number of unnecessary prefetches and leads to severe cache pollution. In contrast, PC_{pre} in GAP often accesses multiple elements within a line with a fixed stride. Consequently, even when all values in the line are extracted, many of them remain useful.

Impact of Source Predictor. Figure 15(b) shows Source Predictor has a particular impact on GAP, whose indirect patterns often involve dependency chains where intermediate operations consume two operands, with one operand being a base array value that remains stable but lies outside the chain. Without Source Predictor, ICP cannot supply these external operands during speculation.

Impact of Demand-Trigger Execution. Figure 15(c) shows that demand-triggered execution plays a secondary role compared to prefetch-triggered execution. The prefetch-triggered execution contributes about 9.02% performance gain over the baseline for SPEC, whereas demand-triggered execution alone contributes about 4.92% for SPEC. The performance benefits enabled by demand-triggered execution stem from two main reasons. First, although the dependency chain from PC_{pre} to PC_{suc} is short (Figure 13), the two instructions can be separated by a large number of control-flow dependent instructions, which may stall issue/execute due to resource contention (e.g., back-end pressure) and significantly extend the effective time gap between PC_{pre} and PC_{suc}. In `gcc`, we observe cases where PC_{suc} executes only after several thousand cycles following PC_{pre}. Second, ICP executes the dependency chain upon receiving the cache-line response of PC_{pre}, which is still earlier than when the core can execute the PC_{suc}. When the core is blocked from issuing subsequent instructions due to resource contention, demand-triggered execution helps improve the prefetching timeliness.

H. Sensitivity Study for Temporal Prefetcher

Sensitivity to prefetch degree. Figure 16(a) studies the sensitivity of DMP+Triangel to the prefetch degree of Triangel. We vary the degree from 1 to 6. The results show that performance remains relatively stable across different degrees, indicating that the hybrid design is not highly sensitive to this parameter. Among the evaluated settings, a degree of 4 achieves the best overall performance, and we use this configuration in our experiments.

Smaller metadata table. Figure 16(b) considers the state-of-the-art temporal prefetcher Streamline with a 0.5MB metadata table. Results show that this setup achieves better per-

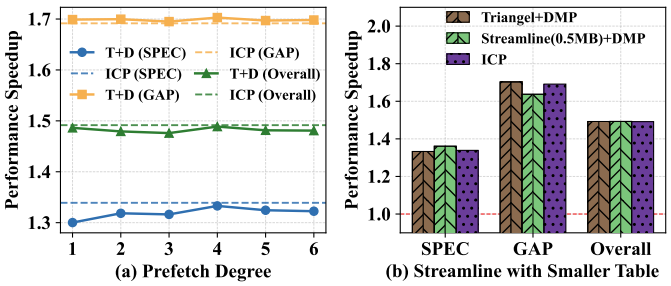

Fig. 16. Sensitivity study for temporal prefetcher degree and table size.

formance on SPEC, due to Streamline's efficient metadata compression, while it degrades on GAP. The degradation also stems from the compressed metadata, which binds multiple consecutive metadata targets with a single lookup address. GAP exhibits few memory-address repetition, making generating multiple prefetches within one lookup less reliable and issue more useless prefetches. Moreover, even when the metadata storage is reduced to 0.5 MB, the overhead remains several orders of magnitude larger than that of ICP.

I. Storage Overhead

TABLE IV
STORAGE OVERHEAD OF ICP.

Structure	Entries	Storage
PC Selector&Classifier	$8 \times 3.875B + 64 \times 1B$	95B
PC Correlation Detector	$128 \times 3.75B + 32 \times 1.75B$	536B
Correlation Table	$64 \times 7B$	448B
Data Extractor	$32 \times 392B$	150B
Source Predictor	$8 \times 9.25B$	74B
Lightweight Calculator	N/A	24B
MSHR Extension	N/A	160B
Commit FIFO Buffer	$8 \times 16B$	128B
ROB Extension	$288 \times 2B$	576B
Total	N/A	2.1KB

Table IV shows ICP's storage overhead. Considering not all ROB implementations explicitly retain source register, we conservatively include the worst-case cost of augmenting the ROB with source register fields. Assuming a 288-entry ROB and an 8-bit register tag, storing two source register tags per entry incurs an additional $288 \times 2 \times 8 = 4{,}608$ bits ($\approx$ 576B) of storage overhead. Two source register fields are sufficient because ICP only needs to model the basic operations supported by the Lightweight Calculator. These operations are at most binary and therefore require no more than two source operands. ICP's SRAM overhead sums to 2.1 KB (17,200 bits) across all structures.

Since 6T SRAM bitcell stores one bit using six transistors, this corresponds to $17{,}200 \times 6 = 103{,}200$ bitcell transistors before accounting for any periphery. To express this in a technology-independent metric, we adopt the common gate-equivalent convention that uses a 2-input NAND (NAND2) [1] as the reference unit. As a result, the bitcells alone map to $103{,}200/4 \approx 25.8$k NAND2 gates. For small SRAM instances, decoders, wordline drivers, and sense amplifiers contribute non-trivial overhead. Therefore, we conservatively scale the bitcell-only estimate by 1.3–1.8$\times$ to account for SRAM periphery, yielding $\sim$33.5k–46.4k NAND2. Finally, adding the remaining non-SRAM logic (e.g., set-associative tag-compare/mux/control and the Lightweight Calculator datapath) results in an overall ICP budget of $\sim$40k–53k NAND2, which corresponds to a 6T-SRAM storage-equivalent capacity of $(40\text{k–}53\text{k}) \times 4/6/8 \approx 3.3\text{–}4.3$ KB.

J. Energy Overhead

We evaluate ICP's energy overhead on memory hierarchy level. Specifically, we use CACTI [38] to model the energy consumption of the on-chip memory hierarchy under 22 nm. For DRAM accesses, we follow the methodology of Triangel [7], assuming the energy cost of a DRAM access is 25$\times$ that of an LLC access. The results show that ICP increases energy consumption by 2.4%, 3.7%, 1.8%, and 7.3% compared to Triangel, Tyche, DMP, and VR, respectively. In contrast, ICP reduces energy consumption by 6.0% and 4.9% compared to DVR and the DMP+Triangel, respectively.

We further evaluate energy overhead introduced by ICP's internal components. The dominant dynamic energy sources within ICP come from (1) accesses to its tables (e.g., Correlation Table) and (2) activations of Lightweight Calculator. To model the per-activation energy of Lightweight Calculator, we follow widely used energy reference points reported by Horowitz [24], which indicate that a simple integer add operation costs 0.1 pJ. Accordingly, we model one Lightweight Calculator operation as 0.1 pJ per activation. For table accesses, [24] also reports that SRAM/cache accesses are at the pJ-level (e.g., $\sim$10 pJ for a 64-bit access to an 8 KB structure). Since ICP's tables are much smaller than KB-scale caches, we bound one table access as 1 pJ. Using measured event counts, we find that internal energy overhead of ICP is negligible, amounting to 0.24% of the memory-hierarchy energy, and therefore does not materially affect overall energy conclusions.

VI. Conclusion

This paper presented ICP, a new hardware prefetching technique that exploits instruction-level correlations to address irregular memory access patterns. Unlike temporal prefetchers that rely on address recurrence, ICP learns stable correlations between instructions, enabling it to prefetch irregular accesses even when memory addresses never repeat. ICP outperforms state-of-the-art temporal prefetchers, indirect prefetchers, and runahead-based solutions, over three orders of magnitude less metadata than temporal prefetchers.

Acknowledgements

We sincerely thank the anonymous reviewers of ISCA'26 for their insightful and constructive feedback. This work is supported by Hong Kong Research Grants Council (RGC) CRF-YCRG C6003-24Y, GRF 16216825, and T46-415/25-R. It was partially conducted by ACCESS – AI Chip Center for Emerging Smart Systems, supported by the InnoHK initiative of the Innovation and Technology Commission of the Hong Kong Special Administrative Region Government.

REFERENCES

[1] "Gate equivalent," https://en.wikipedia.org/wiki/Gate_equivalent.

[2] "SPEC CPU 2006," https://www.spec.org/cpu2006/.

[3] S. Ainsworth, "Ghostminion: A strictness-ordered cache system for spectre mitigation," in *MICRO*, 2021.

[4] S. Ainsworth and T. M. Jones, "Graph prefetching using data structure knowledge," in *ICS*, 2016, pp. 1–11.

[5] S. Ainsworth and T. M. Jones, "Software prefetching for indirect memory accesses," in *CGO*, 2017, pp. 305–317.

[6] S. Ainsworth and T. M. Jones, "Software prefetching for indirect memory accesses: A microarchitectural perspective," *TOCS*, pp. 1–34, 2019.

[7] S. Ainsworth and L. Mukhanov, "Triangel: A high-performance, accurate, timely on-chip temporal prefetcher," in *ISCA*, 2024.

[8] E. S. Alcorta, M. Madhav, S. Tetrick, N. J. Yadwadkar, and A. Gerstlauer, "Lightweight ml-based runtime prefetcher selection on many-core platforms," *arXiv preprint arXiv:2307.08635*, 2023.

[9] J.-L. Baer and T.-F. Chen, "An effective on-chip preloading scheme to reduce data access penalty," in *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, 1991, pp. 176–186.

[10] M. Bakhshalipour, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Domino temporal data prefetcher," in *HPCA*, 2018, pp. 131–142.

[11] M. Bakhshalipour, M. Shakerinava, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Bingo spatial data prefetcher," in *HPCA*, 2019, pp. 399–411.

[12] S. Beamer, K. Asanović, and D. Patterson, "The gap benchmark suite," *arXiv preprint arXiv:1508.03619*, 2015.

[13] R. Bera, A. V. Nori, O. Mutlu, and S. Subramoney, "Dspatch: Dual spatial pattern prefetcher," in *MICRO*, 2019, pp. 531–544.

[14] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH computer architecture news*, 2011.

[15] D. Callahan, K. Kennedy, and A. Porterfield, "Software prefetching," *ACM SIGARCH Computer Architecture News*, vol. 19, no. 2, pp. 40–52, 1991.

[16] M. Chaudhuri, J. Gaur, N. Bashyam, S. Subramoney, and J. Nuzman, "Introducing hierarchy-awareness in replacement and bypass algorithms for last-level caches," in *PACT*, 2012, pp. 293–304.

[17] W. Y. Chen, S. A. Mahlke, P. P. Chang, and W.-m. W. Hwu, "Data access microarchitectures for superscalar processors with compiler-assisted data prefetching," in *MICRO*, 1991, pp. 69–73.

[18] F. Dahlgren and P. Stenstrom, "Effectiveness of hardware-based stride and sequential prefetching in shared-memory multiprocessors," in *HPCA*, 1995, pp. 68–77.

[19] G. Fu, T. Xia, Z. Luo, R. Chen, W. Zhao, and P. Ren, "Differential-matching prefetcher for indirect memory access," in *HPCA*, 2024, pp. 439–453.

[20] G. Fu, T. Xia, M. Yin, P. J. Nair, M. Lis, and P. Ren, "Magellan: A high-performance loop-guided prefetcher for indirect memory access," in *ISCA*, 2025, pp. 601–615.

[21] G. Gerogiannis and J. Torrellas, "Micro-armed bandit: Lightweight & reusable reinforcement learning for microarchitecture decision-making," in *MICRO*, 2023.

[22] E. H. Gornish, E. D. Granston, and A. V. Veidenbaum, "Compiler-directed data prefetching in multiprocessors with memory hierarchies," in *ICS*, 1990, pp. 128–142.

[23] M. He, H. Wang, K. Zhou, K. Cui, H. Yan, C. Guo, and R. He, "Dsdp: Dual stream data prefetcher," in *PACT*, 2022, pp. 372–383.

[24] M. Horowitz, "1.1 computing's energy problem (and what we can do about it)," in *ISSCC*, 2014, pp. 10–14.

[25] I. Hur and C. Lin, "Memory prefetching using adaptive stream detection," in *MICRO*, 2006, pp. 397–408.

[26] Y. Ishii, M. Inaba, and K. Hiraki, "Access map pattern matching for data cache prefetch," in *ICS*, 2009, pp. 499–500.

[27] A. Jain and C. Lin, "Linearizing irregular memory accesses for improved correlated prefetching," in *MICRO*, 2013, pp. 247–259.

[28] S. Jamilan, T. A. Khan, G. Ayers, B. Kasikci, and H. Litz, "Apt-get: Profile-guided timely software prefetching," in *Eurosys*, 2022, pp. 747–764.

[29] S. Jiang, Q. Yang, and Y. Ci, "Merging similar patterns for hardware prefetching," in *MICRO*, 2022, pp. 1012–1026.

[30] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," *ACM SIGARCH Computer Architecture News*, vol. 18, no. 2SI, pp. 364–373, 1990.

[31] T. A. Khan, I. Neal, G. Pokam, B. Mozafari, and B. Kasikci, "Dmon: Efficient detection and correction of data locality problems using selective profiling," in *OSDI*, 2021, pp. 163–181.

[32] J. Kim, S. H. Pugsley, P. V. Gratz, A. N. Reddy, C. Wilkerson, and Z. Chishti, "Path confidence based lookahead prefetching," in *MICRO*, 2016, pp. 1–12.

[33] S. Kim and A. V. Veidenbaum, "Stride-directed prefetching for secondary caches," in *ICPP*, 1997, pp. 314–321.

[34] S. Kondguli and M. Huang, "Division of labor: A more effective approach to prefetching," in *ISCA*, 2018.

[35] M. Li, Q. Zhang, Y. Gao, W. Fang, Y. Lu, Y. Ren, and Z. Xie, "Profile-guided temporal prefetching," in *ISCA*, 2025.

[36] M. Li, Q. Zhang, Y. Ren, and Z. Xie, "Integrating prefetcher selection with dynamic request allocation improves prefetching efficiency," in *HPCA*, 2025.

[37] P. Michaud, "Best-offset hardware prefetching," in *HPCA*, 2016, pp. 469–480.

[38] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0: A tool to model large caches," *HP laboratories*, vol. 27, p. 28, 2009.

[39] A. Naithani, S. Ainsworth, T. M. Jones, and L. Eeckhout, "Vector runahead," in *ISCA*, 2021, pp. 195–208.

[40] A. Naithani, J. Roelandts, S. Ainsworth, T. M. Jones, and L. Eeckhout, "Decoupled vector runahead," in *MICRO*, 2023, pp. 17–31.

[41] A. Navarro-Torres, B. Panda, J. Alastruey-Benedé, P. Ibáñez, V. Viñals-Yúfera, and A. Ros, "Berti: an accurate local-delta data prefetcher," in *MICRO*, 2022, pp. 975–991.

[42] K. J. Nesbit and J. E. Smith, "Data cache prefetching using a global history buffer," in *HPCA*, 2004, pp. 96–96.

[43] S. Pakalapati and B. Panda, "Bouquet of instruction pointers: Instruction pointer classifier-based spatial hardware prefetching," in *ISCA*, 2020.

[44] 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 *HPCA*, 2014, pp. 626–637.

[45] A. Ros and A. Jimborean, "A cost-effective entangling prefetcher for instructions," in *Proceedings of the 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 99–111, entangling instruction prefetcher that selects triggering instruction pairs for timely prefetching. [Online]. Available: https://doi.org/10.1109/ISCA52012.2021.00099

[46] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," *ACM SIGPLAN Notices*, pp. 45–57, 2002.

[47] M. Shevgoor, S. Koladiya, R. Balasubramonian, C. Wilkerson, S. H. Pugsley, and Z. Chishti, "Efficiently prefetching complex address patterns," in *MICRO*, 2015, pp. 141–152.

[48] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Spatial memory streaming," *ACM SIGARCH Computer Architecture News*, pp. 252–263, 2006.

[49] S. Subramaniam, A. Bracy, H. Wang, and G. H. Loh, "Criticality-based optimizations for efficient load processing," in *HPCA*, 2009.

[50] N. Talati, K. May, A. Behroozi, Y. Yang, K. Kaszyk, C. Vasiladiotis, T. Verma, L. Li, B. Nguyen, J. Sun *et al.*, "Prodigy: Improving the memory latency of data-indirect irregular workloads using hardware-software co-design," in *HPCA*, 2021, pp. 654–667.

[51] T. F. Wenisch, M. Ferdman, A. Ailamaki, B. Falsafi, and A. Moshovos, "Practical off-chip meta-data for temporal memory streaming," in *HPCA*, 2009, pp. 79–90.

[52] C. J. Wu, G. H. Loh, M. D. Smith, and D. M. Brooks, "Ship: Signature-based hit predictor for high performance caching," in *MICRO*, 2011.

[53] H. Wu, K. Nathella, M. Pabst, D. Sunwoo, A. Jain, and C. Lin, "Practical temporal prefetching with compressed on-chip metadata," *IEEE Transactions on Computers*, pp. 2858–2871, 2021.

[54] H. Wu, K. Nathella, J. Pusdesris, D. Sunwoo, A. Jain, and C. Lin, "Temporal prefetching without the off-chip metadata," in *MICRO*, 2019, pp. 996–1008.

[55] H. Wu, K. Nathella, D. Sunwoo, A. Jain, and C. Lin, "Efficient metadata management for irregular data prefetching," in *ISCA*, 2019, pp. 449–461.

[56] W. A. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious," *ACM SIGARCH computer architecture news*, vol. 23, no. 1, pp. 20–24, 1995.
[57] 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*, pp. 1–26, 2024.
[58] X. Yu, C. J. Hughes, N. Satish, and S. Devadas, "Imp: Indirect memory prefetcher," in *MICRO*, 2015, pp. 178–190.
[59] P. Zhang, R. Kannan, A. Srivastava, A. V. Nori, and V. K. Prasanna, "Resemble: reinforced ensemble framework for data prefetching," in *SC*, 2022.
[60] Y. Zhang, N. Sobotka, S. Park, S. Jamilan, T. A. Khan, B. Kasikci, G. A. Pokam, H. Litz, and J. Devietti, "Rpg2: Robust profile-guided runtime prefetch generation," in *ASPLOS*, 2024, pp. 999–1013.