Differential-Matching Prefetcher for Indirect Memory Access

Gelin Fu†, Tian Xia†, Zhongpei Luo, Ruiyang Chen, Wenzhe Zhao, Pengju Ren
Xi'an Jiaotong University
{fugelin,luozhongpei,cry20020423}@stu.xjtu.edu.cn,{tian_xia,wenzhe,pengjuren}@xjtu.edu.cn

2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA) | 979-8-3503-9313-2/24/$31.00 ©2024 IEEE | DOI: 10.1109/HPCA57654.2024.00040

***Abstract*—Indirect memory access is a critical bottleneck for modern CPUs, especially for graph analysis and sparse linear algebra applications, where the values of one data array are used to generate the fetching addresses of another array. It often causes irregular data accesses with poor temporal and spatial locality that are difficult to be captured by conventional hardware prefetchers. For many complex workloads, such indirect access patterns may have different types and are nested in a multiple-level form. Moreover, branch mispredictions would further disturb their patterns, making them even harder to detect. As a result, existing hardware prefetchers are unable to fully prefetch complex indirect patterns.**

This paper proposes DMP, a low-cost hardware prefetcher to improve the memory latency in several representative irregular workloads. DMP targets four types of indirect memory access patterns including single, range, multi-level, and multi-way indirect access. DMP uses differential matching to identify an indirect access pattern in pair with its corresponding index stream. Then DMP uses a flexible prefetching mechanism to dynamically adapt the prefetching degree to maintain prefetching coverage.

We evaluate the performance, energy consumption, and transistor cost of DMP among various algorithms from GAP, NAS, and HPCG benchmarks. DMP improves performance by 1.8× (up to 5.6×) on average against state-of-the-art hardware prefetchers and 1.2× (up to 2.3×) speedup against state-of-the-art compiler-based prefetcher Prodigy. Besides, the proposed design is optimized to take only 0.9KB of storage, making it feasible to be integrated into current CPU designs.

I. Introduction

For graph analysis, sparse linear algebra, and many other applications [19], [43], [48], [58], *Memory Wall* is a well-known bottleneck that remains an open challenge for modern CPUs. This bound is mainly caused by irregular memory accesses, where data are arbitrarily accessed with poor locality, resulting in frequent cache misses that cause long latency to fetch data from network-on-chip and DRAM. In these applications, one common irregular memory access pattern is indirect memory access with the basic form of $x[a[i]]$, where an array of data is used as indices to access another data array. Because the values of the index array can be extremely irregular in a wide-range distribution, indirect memory accesses are typically the main cause of the memory bottleneck. Derived from the basic form of $x\big[a[i]\big]$, more complex indirect access patterns can be found in algorithms, including $y\big[x[a[i]]\big]$ where multiple levels of indirection are nested, and multiple parallel indirect accesses using same index such as $x_1\big[a[i]\big] + x_2\big[a[i]\big]$, as well as ranged access for several continuous addresses such as $\big\{x\big[a[i]\big], x\big[a[i]+1\big], x\big[a[i]+2\big]...\big\}$. In some cases, they are combined to form even more complicated indirect accesses, which further deteriorates the *Memory Wall* bottleneck.

Data prefetching is one common technique to alleviate this bottleneck. It predicts future memory access addresses and loads them in advance to hide memory latency [24]. The main principle of hardware prefetcher is to observe the addresses of accesses and try to discover the underlying patterns. Many commercial CPUs deploy stride prefetchers that can detect regular access patterns with fixed address stride [32]. However, such a method is ineffective for indirect access patterns as data are fetched from relatively random addresses. Even for ranged indirect accesses where some data are fetched from several consecutive addresses, the range is typically too short to trigger the stride prefetcher [20]. Therefore, some dedicated hardware prefetchers were proposed to detect and prefetch indirect access patterns of $x\big[a[i]\big]$, such as IMP [75]. However, the actual memory access patterns can be a complicated combination of multiple indirection levels and types, which are difficult to capture. Moreover, recent studies [12], [35], [73], [76] show that graph analysis and sparse linear algebra workloads are likely to encounter many branch mispredictions, which will cause incorrect speculatively-executed instructions for CPUs. This effect will inflate the memory accesses with wrongly speculative memory requests, making it even harder for hardware prefetchers to recognize the indirect access patterns. Recently, there has been an alternate approach to utilizing the compiler to find the indirect access patterns in source code [4], [36], [60], [67], and offload the detected patterns to the hardware prefetching unit during run-time. However, such an approach requires recompilation and has limited compatibility with legacy workloads. In this paper, we focus on the hardware prefetching technique and show that by using our methods, hardware prefetching can achieve equivalent or even superior performance than compiler-based approaches.

To measure the bottleneck of indirect access, we evaluate the performance of various graph analysis and high-performance computation algorithms in Fig. 1. The runtime breakdown in Fig. 1(a) shows that they are all seriously bounded by memory bottleneck. In contrast, the cache miss breakdown in Fig. 1(b) shows that most cache misses are caused by indirect memory accesses even though the CPU has been equipped with stride prefetchers. Fig. 1(c) further breaks down the L2 Cache misses of indirect access into three categories: the basic

†Equal contribution.

2378-203X/24/$31.00 ©2024 IEEE
DOI 10.1109/HPCA57654.2024.00040

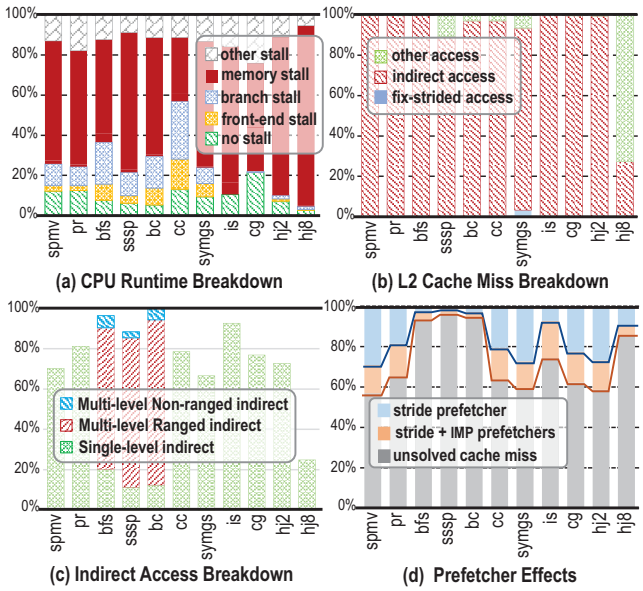


Fig. 1. Performance analysis of multiple graph-analysis and scientific computation algorithms including GAP benchmarks [11] (SPMV, PR, BFS, SSSP, BC, CC), NAS benchmarks [7] (IS, CG), HPCG benchmarks [23] (SYMGS) and Hash-Join [16] (HJ2, HJ8).

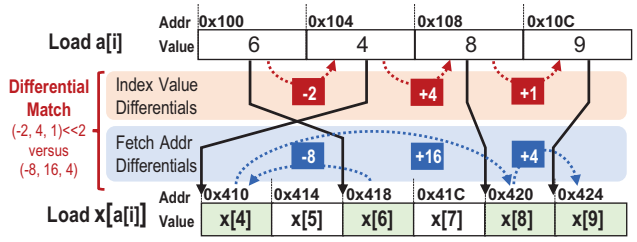


Fig. 2. The fundamental principle of our differential-matching method to detect indirect access patterns by comparing the differential sequences of index values against the differentials of fetched addresses.

single-level indirect access, the multi-level ranged indirect access, and the multi-level non-ranged indirect access. The results show that for graph algorithms of Breath First Search (BFS), Single-Source-Shortest-Path (SSSP), and Betweenness Centrality (BC), the single-level indirection form only takes up small portions ($\leq$ 20%), while most cache misses are due to multi-level ranged indirect accesses, which are difficult to be effectively prefetched. In Fig. 1(d), we depict the effects of using hardware prefetching. It can be observed that using stride prefetcher only reduces a small portion of cache misses. When we combine IMP with stride prefetcher, more cache misses can be removed, but still leaving out most cache misses ($\geq$ 60%) unsolved, especially on BFS, SSSP, and BC, where only $\leq$ 5% cache misses are successfully handled, indicating the limitation of existing hardware prefetching techniques.

Therefore, in this paper, we propose a novel hardware data prefetcher in L1 D-Cache, named Differential-Matching Prefetcher (DMP). Our study is based on two insights. First, the stream-in data usually serves as the initial index for indirect access patterns, so we can leverage the existing stride prefetcher to capture the stream-in indices. Second, though indirect access patterns are difficult to predict, they share the pattern that each fetched data relies on an index value to calculate its offset in the array, so that the differentials of addresses of the fetched data and the differentials of their corresponding index values should be identical or proportional. Derived from these insights, we can start with the stream-in indices captured by the stride prefetcher. To detect the indirect access pattern, we check if the differential sequence of index values matches the differential sequence of indirect access addresses. Fig. 2 depicts the fundamental principle of our method to detect the indirect pattern $x[a[i]]$. Specifically, we observe the load instructions to fetch a and x respectively, tracing the differential sequence of fetched a values as well as the differential sequence of addresses of x. When the sequences are matched with each other, the pattern of $x[a[i]]$ is detected. For ranged indirect accesses of $\{x[a[i]], x[a[i]+1], x[a[i]+2]...\}$, the differential sequence skips the consecutive fetch addresses of x, so that the relationship between index values of a and fetch addresses of x can still be detected. Moreover, for $y[x[a[i]]]$, we can further match the values of x with the addresses of y to detect multi-level indirect access patterns. By iteratively repeating this matching process, we can fully detect complicated indirect access patterns. Furthermore, a dedicated prefetching unit with an adaptive prefetching degree is proposed to generate efficient prefetch requests for complicated indirect access patterns.

We evaluate DMP with extensive experiments on various benchmarks and workloads. The results indicate that DMP can detect most of the indirect access patterns in complicated algorithms, resulting in a 1.8× speedup on average against state-of-the-art hardware prefetchers and a 1.2× speedup on average against the best compiler-based prefetcher. Besides, DMP is optimized to use only 0.9KB of storage, making it feasible to implement. Our contributions are as follows:

- We propose a new hardware prefetcher named DMP, which efficiently detects and prefetches multiple-level and ranged indirect access patterns. An adaptive degree mechanism is proposed to learn the appropriate degree for ranged access.
- We propose several mechanisms of sample window, filtering, and retry to to fast detect the indirection patterns and to alleviate the interference of branch misprediction and out-of-order execution.
- We provide a low-overhead implementation of DMP in hardware and demonstrate its practicality by using RTL to evaluate the hardware cost and energy of DMP.
- We extensively evaluate DMP and show it outperforms the state-of-the-art prefetchers over a wide variety of workloads. Extensive discussions are made to study the effects of DMP.

II. Preliminary

A. Indirection Types

1) Single and Ranged Indirection: Depending on how an index is used to fetch data, indirect accesses can be categorized

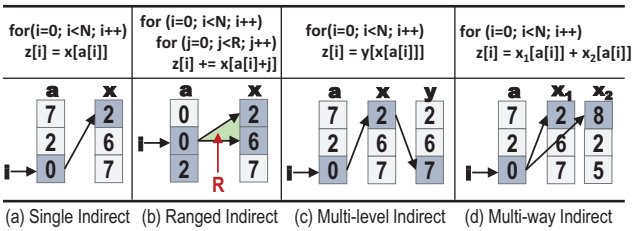


Fig. 3. Examples of various indirect access types include single, ranged, multi-level, and multi-way indirection patterns.

as single and ranged indirection. Single indirection is the basic type of indirect accesses, where each index value is used as the offset to fetch one data at a time, as depicted in Fig. 3(a). In contrast, for ranged indirection, each index is used to fetch from consecutive addresses, as shown in Fig. 3(b). The scale of range could be dynamic or fixed, depending on the algorithms. For example, for the BFS algorithm, the range equals the number of vertex neighbors which is variable during the process, while for sparse matrix-matrix multiplication (SPMM) the range is the fixed width of the dense matrix. In our design, an adaptive prefetching degree is used to determine the rational range to prefetch for different algorithms and workloads.

2) Multi-level and Multi-way: In many algorithms, multiple indirect access patterns co-exist in combinations, which can be classified as two forms of multi-level and multi-way indirection. Multi-level indirection means that several indirect access patterns are nested so that the fetched data of one indirect access can be used as the index of another indirect access, as shown in Fig. 3(c). The depth of levels may vary in different algorithms. In this case, only the first level is using stream-in indices while the deeper levels are using indices from arbitrary positions. On the other hand, the multi-way indirection means that several indirect accesses share the same index value as their addressing offsets, as depicted in Fig. 3(d).

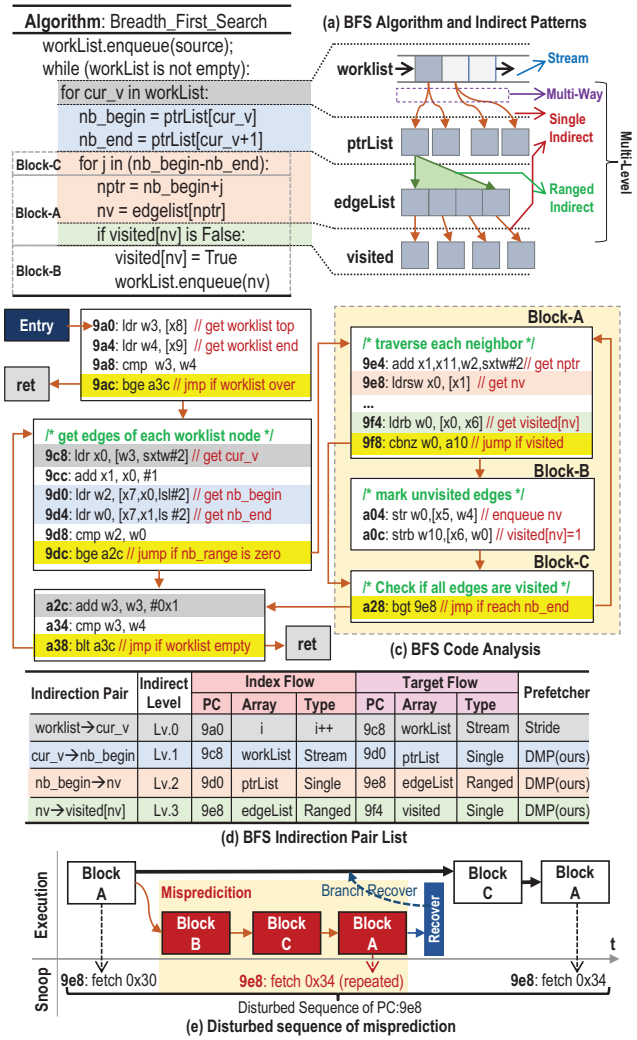
The different single/ranged types and multi-level/multi-way types can further combine to form quite complicated indirect patterns. To give an example, we list the pseudo-code of the Breadth First Search algorithm in Fig. 4(a) and depict its indirection access patterns in Fig. 4(b). We can observe three-level nested indirect access patterns of both single and ranged types, which are very difficult for hardware prefetchers to detect and prefetch efficiently. For algorithms such as SSSP and BC, their indirect accesses are even more complex with deeper indirect levels and several multi-way patterns.

B. Indirection Pair

To efficiently represent the complex indirection access patterns, we define the *indirection pairs* as:

$$pair(PC_{index}, Type_{index}, PC_{target}, Type_{target}) \quad (1)$$

where PC_{index} is the program counter of instruction that loads index values, and PC_{target} represents the load instruction of target data whose fetching addresses are calculated based on the values fetched by PC_{index}. $Type_{index}$ and $Type_{target}$ represent whether the indices and target data are loaded as



Indirection Pair	Indirect Level	Index Flow			Target Flow			Prefetcher
		PC	Array	Type	PC	Array	Type	
worklist→cur_v	Lv.0	9a0	i	i++	9c8	workList	Stream	Stride
cur_v→nb_begin	Lv.1	9c8	workList	Stream	9d0	ptrList	Single	DMP(ours)
nb_begin→nv	Lv.2	9d0	ptrList	Single	9e8	edgeList	Ranged	DMP(ours)
nv→visited[nv]	Lv.3	9e8	edgeList	Ranged	9f4	visited	Single	DMP(ours)

(d) BFS Indirection Pair List

Fig. 4. Example code analysis of Breadth First Search (BFS) algorithm for indirect access patterns with the list of indirection pairs. A disturbed memory access sequence due to branch mispredictions is also provided.

single or ranged indirect access. In multi-level indirection, several indirection pairs can be chained up as long as one pair's PC_{index} equals the other pair's PC_{target}. To give an example, Fig. 4(c) depicts the assembly code of BFS algorithm and Fig. 4(d) lists the corresponding indirection pairs. By using our differential-matching method, we can identify the indirection pairs among indirect access instructions (*0x9c8*, *0x9d0*, *0x9e8*) and the stream-in load instruction (*0x9a0*), building a chain of indirection pairs. With this knowledge, a prefetching chain can be initiated by observing the values fetched by one pair's PC_{target} and using them as indices of the successor pair, which allows us to prefetch deep-level indirect accesses at a very early time. Meanwhile, $Type_{index}$ and $Type_{target}$ are used to facilitate the prefetching. For an indirection pair whose $Type_{index}$ is ranged, we observe several continuous index values fetched by PC_{index} and use each of them to generate a prefetch address. On the other hand, when $Type_{target}$ is ranged, multiple prefetching addresses are generated for each

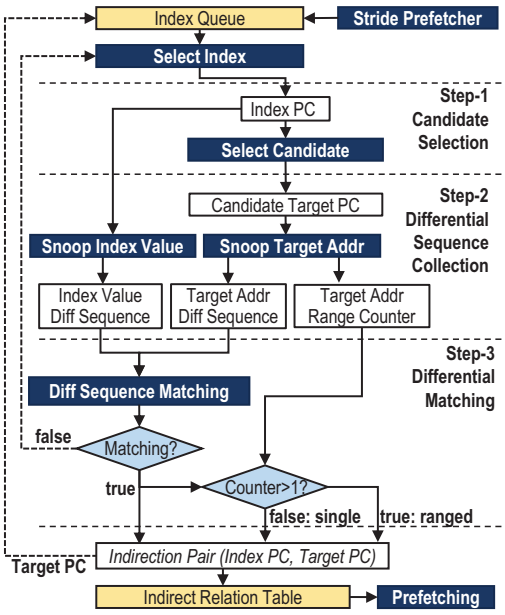


Fig. 5. Indirect access pattern detection flow.

index value. In this way, both single and ranged indirect accesses can be efficiently and properly handled.

C. Interference of Branch Mispredictoins

Hardware prefetcher snoops the cache port to observe the data accesses. To build indirection pairs, our approach relies on the observation of the correct sequence of addresses and values fetched by the PC_{target} and PC_{index} instructions. However, on modern out-of-order CPUs, irregular workloads are likely to trigger a large number of branch mispredictions that disturb the sequence of load instructions. This is because branch mispredictions cause the CPU to abort speculatively executed instructions and restart from the branch point, which may cause repeated execution of certain indirect access instructions. Such repeated access may disturb the sequence being observed by the prefetcher. For example, Fig. 4(c) denotes BFS's inner loop, which is composed of three basic blocks *Block-A/B/C*. *Block-A* ends with a branch that goes to *Block-B* if the subject edge has not been visited, or otherwise jumps to *Block-C*. However, this branch is hard to predict as its condition is dynamically changing. When this branch is mispredicted, the CPU follows the execution flow depicted in Fig. 4(e). We can observe that *Block-A* is executed once on the speculative path and then repeated after the misprediction is fixed, which results in the instruction *PC=0x9e8* issuing repeated memory requests on the same address. Such a disturbed sequence will interfere with the observation and detection of prefetchers, which is a critical but usually overlooked problem for hardware prefetchers. Therefore, we propose that by filtering repeated values in the observed sequence, the interference of branch mispredictions can be effectively mitigated.

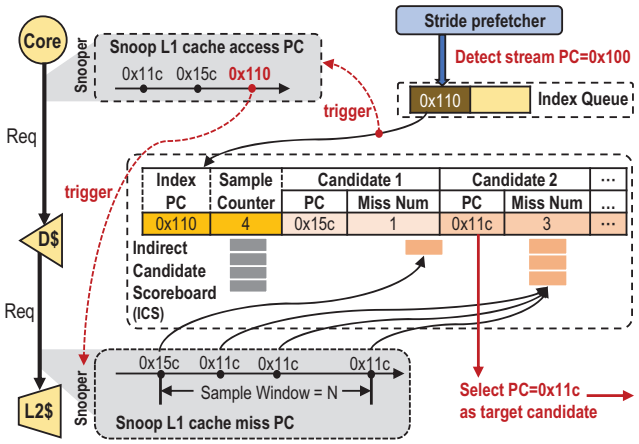


Fig. 6. Candidate selection.

III. DMP: A Differential Matching Prefetcher

Differential Matching Prefetcher (DMP) is a data prefetcher dedicated to indirect memory accesses, which is sited at L1 D-cache where it observes all memory requests generated by the CPU to detect indirect access patterns. Then it generates prefetch requests according to the detected patterns and the observed index values. This section explains the detection and prefetching mechanism of DMP.

We first introduce the detection mechanism. Fig. 5 depicts the overall detection flow. As shown, DMP uses the *Index Queue* structure to hold the instruction PCs that are likely to fetch indices in indirect access patterns. Such instructions include the PCs found by the stride prefetcher, as most indirect access patterns begin with stream-in indices such as $a[i]$. For multi-level indirect patterns, the target PC (PC_{target}) in an indirection pair should also be considered as potential index PC for another indirection pair and thus is also included in *Index Queue*. To find an indirection pair, DMP selects an index PC (PC_{index}) from *Index Queue* and attempts to find its corresponding target PC (PC_{target}) within a three-step process. First, for the index PC, DMP selects the instruction that is likely to be the target PC, termed as the target candidate PC. Then, DMP collects differential sequences for the index PC and target candidate PC by snooping their memory requests at the cache ports. Finally, DMP checks for the indirection pair by performing differential matching between the index PC and target candidate PC. At successful matching, an indirection pair is identified and then stored to be used for prefetching. Meanwhile, to find deeper indirection levels, the target PC can be used as an index PC to initiate further detection process, as shown in Fig. 5. In the following sections, we first introduce the detection steps and then explain how the detected patterns are used for the prefetching.

A. Candidate Selection

As shown in Fig. 5, DMP starts the detection process by selecting an index PC from *Index Queue*. The goal of candidate selection is to find the potential memory access instructions that use the index PC to generate their fetching addresses,

which are termed target candidate PCs. Since there are a huge number of memory access instructions on the fly, DMP needs to narrow down the candidates by selecting only the instructions most likely to be the target PC. For this purpose, we select candidates following several assumptions. First, the target PCs are likely to trigger more cache misses since the access addresses of target PCs are very irregular. Second, memory requests of the target PC are likely to be issued soon after the corresponding index PC.

Based on these insights, we propose an efficient candidate selection scheme, as described in Fig. 6. In the given example, an instruction (*PC:0x110*) captured by the stride prefetcher is recorded in the *Index Queue* and is used as the index PC. To find its corresponding target PCs, DMP places the index into the Indirect Candidate Scoreboard (ICS), which meanwhile triggers DMP to snoop the L1 D-Cache access port to check for the index PC. Whenever the index PC is encountered, DMP starts to snoop the following N D-Cache misses and records their PCs and cache miss numbers in ICS. The number N is termed as *sample window*, which is used to make sure only memory accesses close to the index PC access are concerned. For each instruction encountered in the *sample window*, ICS records its PC and cache miss number. When the total N cache misses have been observed, the *sample window* ends, and the PC with the most miss number in ICS is selected as the target candidate PC for the subject index PC.

In practice, DMP sets a 64-length sample window, and the maximal number of candidates in each ICS entry is 16.

B. Differential Sequence Collection

To determine whether a target candidate PC is the corresponding target PC for an index PC, DMP collects the sequences of the index PC's fetched values and the target candidate PC's access addresses by snooping L1 D-Cache access ports. The two sequences are then compared using the differential-matching method.

Fig. 7 depicts the schemes of differential sequence collection. For both index PC and target candidate PC, DMP first eliminates the interference of branch mispredictions which may cause repeated memory accesses. As shown in Fig. 7, the snooped sequences always go through a *Repetition Filter* before they are collected. This filter removes consecutive elements that have the same value so that the repeated accesses caused by wrong speculations are removed. Our experiments have shown that this filer is adequate to eliminate most interference caused by branch mispredictions.

Fig. 7(a) depicts the sequence collection of index PC. For a given index PC, DMP snoops the response port of L1 D-Cache to observe the loaded data. After *Repetition Filter*, the sequence of loaded data values is processed to generate the *index value differential sequence* by calculating each fetched value with its previous value. The resulting sequence is saved in a FIFO buffer in the *Index Table* (IT). Meanwhile, the latest index value is also recorded in IT. Whenever a new fetched value of index PC is observed, DMP inserts its differential with

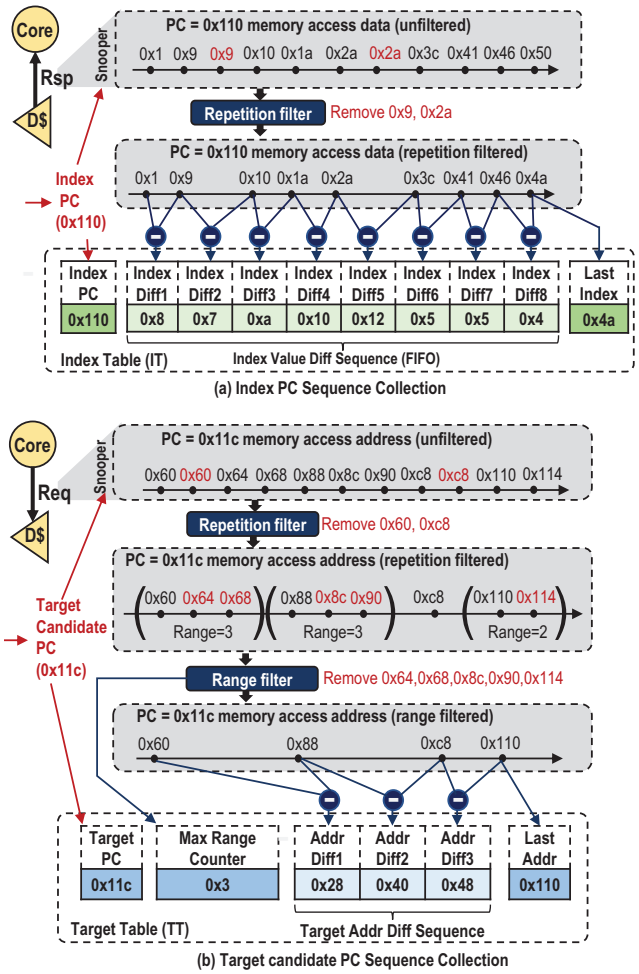


Fig. 7. Differential sequence collection.

the previous value into the *index value differential sequence* and updates the latest index value in IT.

Fig. 7(b) depicts the sequence collection of target candidate PC. DMP snoops its request addresses at the L1 D-Cache access port, and the observed sequence also first undergoes the *Repetition Filter*. Considering the possibility of ranged indirection where one index value is used to generate several consecutive addresses of the target PC, we propose a *Range Filter* to further process the address sequence to remove ranged addresses. As shown in Fig. 7(b), for each range, *Range Filter* removes the consecutive addresses except the first element in the range, so that the filtered target address sequence maintains its correspondence with the index value sequence. *Range Filter* also counts the scales of ranges and records the maximal encountered range scale in the *Target Table* (TT), which is used to determine single and ranged types in later steps. After *Repetition Filter*, the resulting sequence is processed to generate the *target address differential sequence* that is saved in TT along with the latest address of the target candidate PC. It should be noted that the index sequence is always ahead of the target sequence because indirect memory access must be

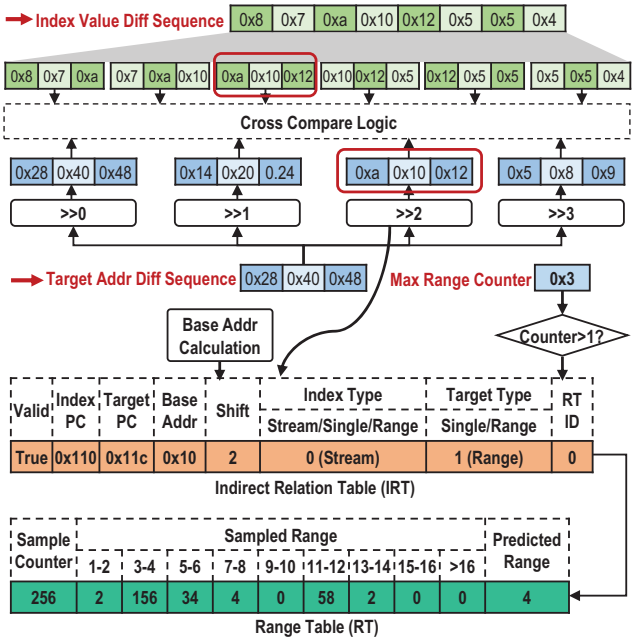


Fig. 8. Differential matching.

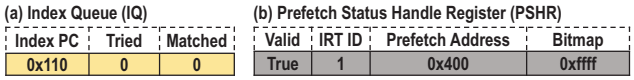


Fig. 9. Several tables and register structures used in DMP.

issued after the acquisition of the corresponding index value. Therefore, unlike the *index value differential sequence* which always updates with new index values, the *target address differential sequence* only saves a fixed number of candidate address differentials. Whenever this sequence gets full, it triggers the differential matching step.

C. Differential Matching

This differential matching step is to check if the index PC and target candidate PC are indirection pairs. As shown in Fig. 8, we slide the *target address differential sequence* on the *index data differential sequence* and compare their values. To represent 8/16/32/64-bit data formats for target addressing, values in the *target address differential sequence* are right-shifted with 0/1/2/3 bits, respectively. On a successful match, a valid indirection pair is detected and a new entry of the Indirect Relation Table (IRT) is allocated to hold the index PC and the target PC. The IRT entry also needs to record the corresponding shift number as well as the base address of the target data structure, which can be calculated as:

$$BaseAddr = LastAddr - (LastIndex - \sum_{l \to m} IndexDiff(i)) << shift \quad (2)$$

where *LastAddr* and *LastIndex* are the last observed address and index values stored in IT and TT, and $\sum_{l \to m} IndexDiff(i)$ accumulates the differentials between the last observed stream data and the matched data recorded in IT. Additionally, DMP detects the indirection type of target PC by checking the*Max Range Counter* in the TT. If the value equals one, it is a single indirection. Otherwise, the indirection type is ranged. The detected type is also recorded in the IRT entry.

Moreover, we need to determine the number of consecutive elements to prefetch for the ranged access. As the lengths of ranged accesses are dynamically influenced by input data distribution and algorithms, we propose the *adaptive degree prediction* mechanism to adjust prefetching degrees on the fly. Specifically, whenever an indirection pair is detected, we allocate an entry of the *Range Table* (RT) for the new target PC. Then DMP snoops the memory accesses of this target PC and uses the *Range Filter* to count for the length of ranges, which are recorded in RT, as shown in Fig. 8. With these historical records, the most frequently encountered range length is used as the predicted range for this PC. Whenever the subject PC is used for generating the prefetch requests, the predicted range saved in RT is used as the prefetching degree.

D. Scheduling of Detection

To detect multi-level and multi-way patterns, DMP supports the simultaneous detection of multiple indirection pairs by providing multiple ICS/IT/TT entries. Moreover, DMP uses the *Index Queue* to holds all the stream PCs and already-detected index flow PCs. Whenever an indirection pair is detected, its target PC is added into IQ as a potential index PC so that deeper indirection levels can be searched.

Meanwhile, in complex algorithms where different indirect accesses are mixed up, the detection of an indirection pair may not be successful with only a one-shot try as the observed sequence may be heavily noised. Moreover, by using filters to alter the snooped sequence, we cause the risk of missing in-direction pairs which should have been successfully detected. Therefore, to fully detect the entire indirect patterns, DMP uses a retry mechanism to perform repeated detection. As shown in Fig. 9(a), for each potential index PC saved in IQ, its detection history is recorded, including how many times they have undergone the detection process (*Tried*), and how many indirection pairs have been found (*Matched*). DMP always selects the index that has the highest value of $\frac{Matched+1}{Tried}$, which equals ∞ if *Tried=0*. This provides an explore-exploit strategy as firstly non-tried index PCs are detected and then the ones that have found target flows are encouraged to try more because we assume they have the potential to have more indirect patterns.

E. Prefetching Generation

Generally, DMP prefetches indirection pairs in a prefetching chain, where the predecessor pair's target PC's loaded values are the successor pair's index PC values that are then used to calculate the successor pair's target PC access addresses. An exception is the first indirection level, whose index PC is prefetched by the stride prefetcher. DMP's prefetching requests are handled in the Prefetch Status Handling Register (PSHR). As shown in Fig. 9(b), PSHR records the prefetching address and the corresponding IRT entry ID. It also holds a bitmap to indicate the outstanding elements in the requested cache block. To prefetch the ranged indirect pattern where a group

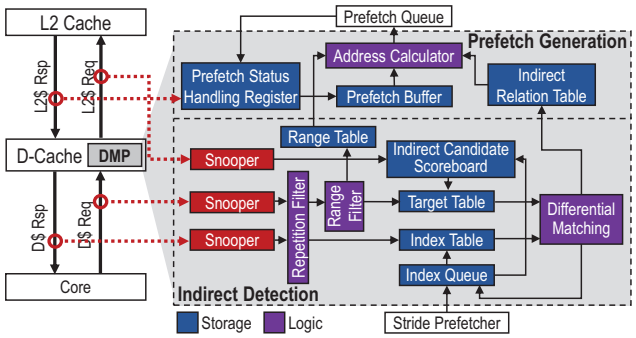


Fig. 10. Overview of DMP hardware design.

of consecutive addresses is requested, DMP looks up the RT to find the predicted range for this PC and sets the corresponding consecutive bits in the PSHR bitmap. Whenever a request in PSHR is finished, DMP receives the response cache block and uses the elements marked in the PSHR bitmap as index values to generate the target PC's prefetching addresses, by using the *Base Address* and *Shift* stored in the corresponding IRT entry. The predicted range in the corresponding RT is also used if the target PC is ranged type.

Meanwhile, though the response data of PSHR are the index values that are used for prefetching, these data are not suitable to be stored in the L1 D-Cache. This is because, for multi-level or multi-way indirection patterns, the amount of prefetched data tends to be large which may pollute the L1 D-Cache and decrease the performance. Therefore, DMP uses a prefetch buffer to hold the prefetched index values for the calculation of prefetching addresses, while the cache blocks of prefetched data are actually placed in L2 Cache instead of in L1 D-Cache.

IV. Hardware Design and Implementation

A. Hardware Design

Fig. 10 shows the components of DMP organized as two parts: indirect detection and prefetch generation. The indirect detection includes detection tables and differential modules. Snoopers are inserted to monitor L1 D-Cache accesses including request PC, address, and response data. Meanwhile, prefetch generation is attached to the L1 D-Cache and is responsible for generating indirect prefetches. As shown in Fig. 10, DMP monitors the L1 D-Cache response of the Index PCs and finds information about indirect patterns of this Index PC in the IRT, Then DMP generates the prefetch request for this Index PC using schemes described in Section III-E and the prefetching address is translated to a physical address using the TLB and issued to L2 Cache.

We provide four entries in ICS/IT/TT tables to allow four indices to be simultaneously detected, while IRT has 15 entries to handle a maximum of 15 indirection pairs. RT has four entries so that DMP can handle a maximum of four ranged indirect patterns in parallel. Each entry in the prefetch queue has a corresponding IRT ID to transfer this information to PSHR. Table I shows the table size of DMP, which requires only 912B (0.9KB) storage.

TABLE I
Storage Overhead of DMP

Table	Entry Size (bit)	Entry Number	Total Size (byte)
ICS	290	4	145
IQ	18	8	18
IT	300	4	150
TT	140	4	70
IRT	64	15	120
RT	92	4	46
PSHR	50	12	75
Prefetch Buffer	32	64	256
Prefetch Queue	4	64	32
Total			**912**

TABLE II
Simulation Parameters of Baseline System

Main Core	
Core	4-Wide, out-of-order, 2.5Ghz
Pipeline	224-entry ROB, 128-entry IQ, 72-entry LQ, 56-entry SQ, 180 Int registers, 168 FP registers, 4 Int ALUs, 2 FP ALUs, 2 Mult/Div ALUs
Branch Pred.	TAGE Predictor [63], 4096-entry BTB
Memory	
L1 I-Cache	32KB, 8-way, 2-cycle hit lat, 16MSHRs
L1 D-Cache	32KB, 8-way, 4-cycle hit lat, 16MSHRs
L2 Cache	256KB, 4-way, 12-cycle hit lat, 32MSHRs
Memory	DDR3-2133,tRP:13ns,tRCD:13ns,tRAS:33ns

B. Implementation

To evaluate the hardware cost of our design, we implement a register-transfer-logic (RTL) version of our design using *SystemVerilog*. Implementation includes mainly four full functional components: ICS, IT, TT, and IRT. We synthesize our implementation using Synopsys Design Compiler [66] with 28nm library from TSMC [69] to estimate DMP's overhead and the result shows that DMP consumes 0.018mm^2 of area and 17mW of dynamic power. Among those tables, IT occupies the largest area which is about 36%. Considering a typical 28nm 32KB Data Cache which costs 0.273mm^2 and 315mW, DMP takes 6.8% area and 5.4% power. DMP also costs much less resources when compared with the state-of-the-art prefetcher Pythia [14] which consumes 0.33mm^2 of area and 55.11mW of power. Besides, Compared with Intel Core i3-6006U [29], a low-end 2-core desktop-class Skylake processor with 60W TDP and Intel Xeon Platinum 8180M [31], a high-end 28-core server-class Skylake processor with 205W TDP, multi-core configuration of DMP gets 0.07% and 0.03% area overhead as well as 0.23% and 0.05% power overhead respectively. We can conclude that DMP only brings negligible hardware overhead to real processor design.

V. Evaluation

A. Methodology

We use the event-driven GEM5 simulator [45] to perform evaluations. We simulate an Intel Skylake microarchitecture [30] and Table II provides the key micro-architecture parameters of the simulated CPU. The baseline configuration is a non-prefetching scheme. To profile the performance of the key parts of these algorithms, we use the region-of-interest

(ROI) utility of GEM5 to ignore initialization cost, including reading graphs from files and populating data structures.

Prefetchers: We compare the performance of DMP against six state-of-the-art prefetchers including Ainsworth's software prefetching [3], IMP [75], IPCP [59], Berti [57], Event-trigger [4] and Prodigy [67]. Table III shows the parameters of all prefetchers including software (SW), hardware (HW), and software-hardware (SW-HW) co-designed prefetchers. We use the artifacts provided by the authors for evaluation.

Workloads: We evaluate DMP using twelve memory-bound irregular workloads from different fields including graph, database, and HPC, as described in Table IV. Specifically, sparse matrix-vector multiplication (SPMV), page rank (PR), breadth-first search (BFS), single-source shortest path (SSSP), betweenness centrality (BC), and connected components(CC) are graph algorithms from GAP benchmark [11]. Symmetric Gauss-Seidel Smoother (SYMGS) and sparse matrix-matrix multiplication (SPMM) are from HPCG benchmark [23] and TACO compiler [42] as representative sparse linear algebra applications. Integer sort (IS) and conjugate gradient (CG) are from NAS parallel benchmark [7] as computational fluid dynamics applications. Hash Join [16] (two elements per bucket for HJ2 and eight elements per bucket for HJ8) is a popular database kernel. As listed in Table IV, different workloads have different types of indirect access patterns including *Single*, *Range*, *Multi-level* (M-L), and *Multi-way* (M-W). As for datasets, we use real-world graph data collection [22] for graph algorithms, as shown in Table V. CLASS-B (33M) scale is tested for IS and CG. For HJ2 and HJ8, we use *-r=12800000* and *-s=12800000* as input parameters.

B. Prefetching Metrics

a) Prefetching Coverage: This metric measures the fractions of data transfer that can be covered by prefetch requests. Fig. 11(a) shows the prefetching coverage of the prefetchers on all benchmarks. For all benchmarks, DMP outperforms IMP and Berti with great margins, reaching an average coverage of 79.5% (up to 99%). It is noteworthy that IMP and Berti are ineffective for graph algorithms such as BFS/SSSP/BC, while DMP achieves a high level of coverage in these applications by utilizing range indirect detection and adaptive degree generation approaches.

b) Prefetching Accuracy: This metric measures how accurately the prefetcher can predict memory access addresses. As shown in Fig. 11(b), the accuracy of DMP is much higher than other approaches on all benchmarks, achieving an average accuracy of 92.8% (up to 100%). Especially for graph search applications such as BFS, SSSP and BC, benefiting from accurate and fast detection schemes, DMP reaches more than 80% accuracy while for IMP and Berti most prefetches are incorrect because they cannot detect most indirect patterns.

c) Prefetching Timeliness: Late prefetching occurs when a prefetch request has been issued but not returned when its data is required by CPU, in which cases the memory access latency is not fully hidden. Fig. 11(c) shows the results, indicating that both DMP and IMP perform very well in terms

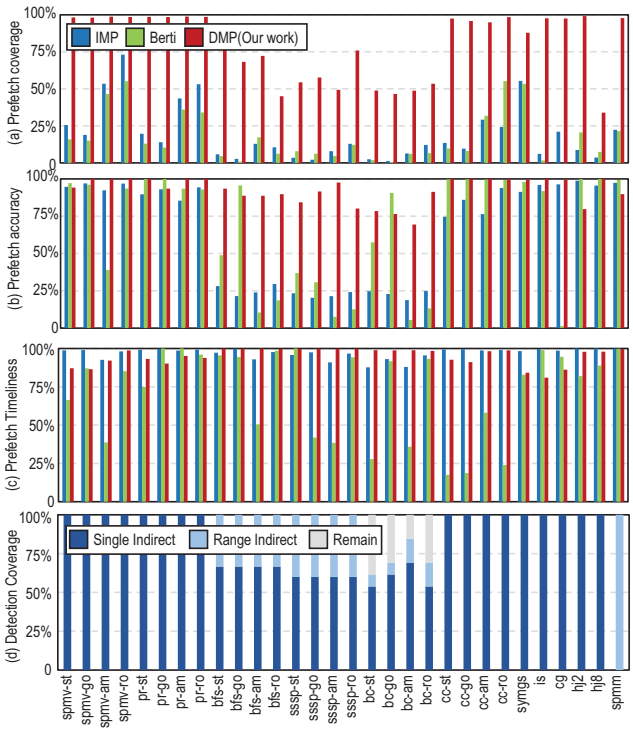


Fig. 11. Prefetching metrics for DMP and compared prefetchers.

of timeliness on most applications, with DMP reaching an average timeliness of 95.2% (up to 99.9%).

d) Detection Coverage: This metric measures if a prefetcher can detect all indirect patterns in benchmarks. To evaluate this metric, we manually analyze all the benchmark kernel codes and count all the indirect patterns as ground truth. Then we examine whether the tested prefetcher can find all indirect patterns. Fig. 11(d) depicts the proportion of indirect patterns that are detected by DMP for different benchmarks, while different types of indirect accesses are listed, including single and ranged indirection as well as the remaining patterns that aren't successfully detected. The results show that DMP's average detection coverage can reach 85.4% (up to 100%), indicating that DMP can effectively detect indirect accesses in all the benchmarks. Moreover, DMP can always correctly recognize the indirection types.

C. Overall Performance

Fig. 12 and Fig. 13 compare DMP with other hardware prefetchers and compiler-assisted prefetchers respectively, with the baseline of non-prefetching. DMP achieves a maximum speedup of 5.6× for IS and an average speedup of 2.06×.

Both IPCP [59] and Berti [57] are state-of-the-art general-purpose hardware prefetchers. which generate prefetch addresses based on learning from history memory access sequences and perform well on regular and trainable memory access patterns. However, they cannot achieve good performance gains on irregular memory access sequences. We observe that the speedups of IPCP and Berti are less than 1× for a few

TABLE III
CONFIGURATION OF EVALUATED PREFETCHERS

Prefetcher	Configuration	Category	Storage Overhead
DMP(our work)	4-entry ICS, 4-entry SDDT, 4-entry IADT, 16-entry IRT	HW	0.9KB
IMP [75]	16-entry PT, 4-entry IPD	HW	0.7KB
IPCP [59]	128-entry IP Table, 8-entry RST Table, 128-entry CSPT Table	HW	0.9KB
Berti [57]	128-entry History table, 16-entry Table of deltas, 16-bit timestamp per entry	HW	2.5KB
Event-Trigger [4]	40-entry observation queue, 12 PPUs:4-stage in-order pipeline, 1Ghz	SW-HW	8.0KB
Prodigy [67]	11-entry node table, 14-entry edge table,15-entry PFHR	SW-HW	0.8KB
Software Prefetch [3]	Software prefetching for indirect access	SW	/

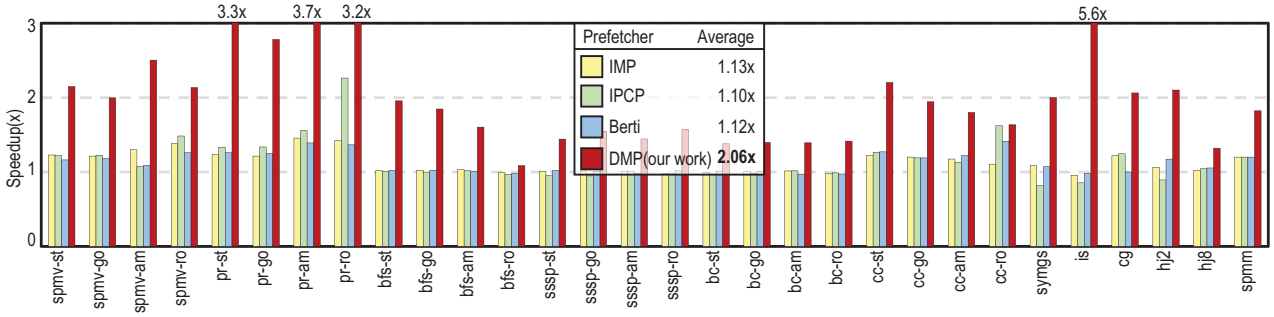


Fig. 12. Execution time speedup provided by DMP and compared hardware prefetchers over the non-prefetching baseline.

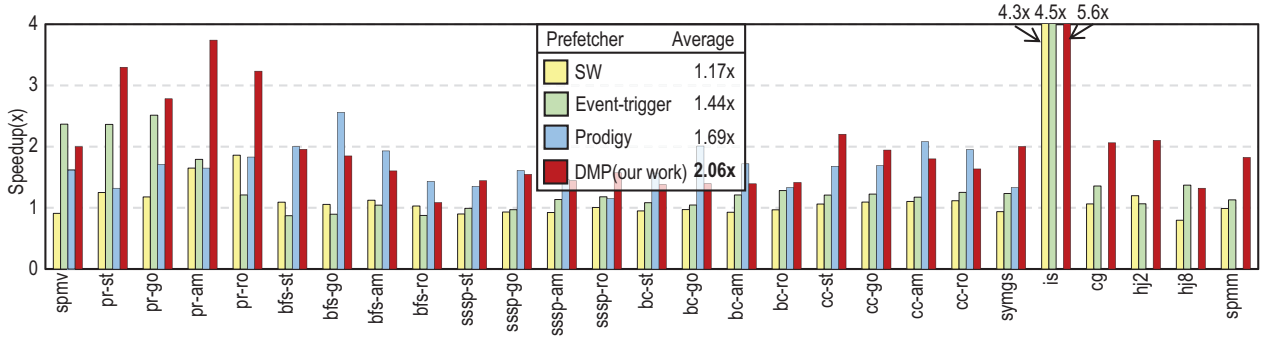


Fig. 13. Execution time speedup provided by DMP and compared compiler-assisted prefetchers over the non-prefetching timeline. We do not report the performance of Prodigy on IS,CG,HJ2,HJ8,SPMM because of simulation faults.

TABLE IV
KERNEL CONFIGURATIONS

Benchmark	Source	Indirect Access Pattern			
		Single	Range	M-L	M-W
SPMV	GAP [11]	✓			
PR	GAP [11]	✓			
BFS	GAP [11]	✓	✓	✓	
SSSP	GAP [11]	✓	✓	✓	✓
BC	GAP [11]	✓	✓	✓	✓
CC	GAP [11]	✓			
HJ2	Hash-join [16]	✓			
HJ8	Hash-join [16]	✓			
IS	NAS [7]	✓			✓
CG	NAS [7]	✓			✓
SYMGS	HPCG [23]	✓			
SPMM	TACO [42]	✓	✓	✓	

TABLE V
REAL-WORLD DATASETS FOR EVALUATION

Matrix	Vertices	Edges	Domain
web-Stanford	281,903	2,312,497	Web Graphs
web-Google	916,428	5,105,039	Web Graphs
amazon-2008	735,323	5,158,388	Co-purchase
RoadNet-PA	1,090,920	3,083,796	Roads

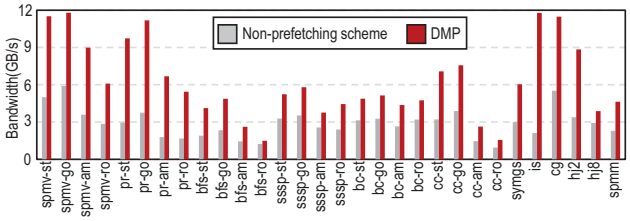


Fig. 14. Memory bandwidth for DMP and non-prefetching baseline. The peak memory bandwidth is 12.8GB/s.

workloads. Because they mistake indirect accesses for other kinds of patterns, resulting in many incorrect prefetch requests.

IMP [75] is a hardware prefetcher for indirect memory access. Compared to IMP, DMP achieved 1.82× speedup. This is because IMP fails to detect indirect patterns due to out-of-order execution, and is not suitable for more complicated indirect access patterns (e.g., ranged, multi-level, multi-way).

For software prefetching, DMP achieves 1.76× speedup

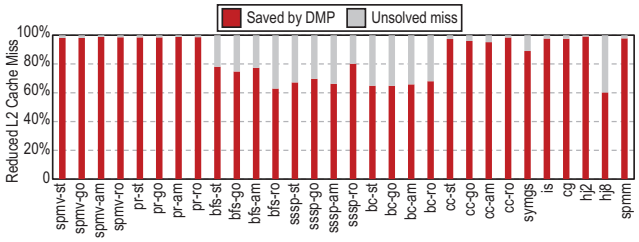


Fig. 15. Percentage of L2 Cache miss reduced by DMP. Red is better.

over the method proposed by Ainsworth [2]. Experiments show that software prefetching is effective in improving pipeline efficiency and increases the CPU IPC by 1.7×. However, this method also incurs additional instruction overhead, increasing the number of instructions by an average of 1.49× over the non-prefetching baseline, which undermines its benefits.

We also evaluated the performance of the latest hardware-software co-designed prefetchers, including Event Trigger [4] and Prodigy [67]. DMP provides a speedup of 1.43× on average over Event-trigger. For Prodigy, we fail to reproduce the performance of IS, CG, HJ2, HJ8, and SPMM because of simulation faults. So we only consider the performance of the rest workloads and DMP achieves a speedup of 1.15× on average. We observe that DMP performs better than Prodigy in most applications. But Prodigy outperforms DMP on some workloads including BFS, SSSP, and, BC. This can be attributed to two factors. First, DMP's detection coverage is not always 100% because some indirect access patterns may not be successfully detected, whereas Prodigy can perfectly match all indirect access patterns through compilation. Second, while DMP incorporates adaptive degree prediction for prefetching, it still falls short compared to Prodigy, which benefits from static compile-time information, particularly in range prefetching. It is worth to be noted that as a hardware-software co-designed prefetcher, Prodigy requires programmer annotations and compiler analysis to specify the indirect access relationships during compilation. This necessitates a deep understanding of program data structures and access patterns by the programmer. On the other hand, DMP, as a pure hardware prefetcher, can perform prefetching by analyzing the indirect access relationships without any modification to the source code, offering higher compatibility.

To further evaluate DMP's effect on memory optimization, Fig. 14 depicts the DDR memory bandwidth usage of DMP, which greatly boosts bandwidth, reaching an average of 2.1× bandwidth than the non-prefetching baseline. We also show L2 cache miss reduction by DMP In Fig. 15, where L2 cache misses are reduced by 77% than non-prefetching. Therefore, with highly-accurate detection of most indirect patterns, while ensuring most prefetched data are useful, DMP greatly improves memory access efficiency, which explains DMP's significant speedup from the view of the computer system.

D. Discussions

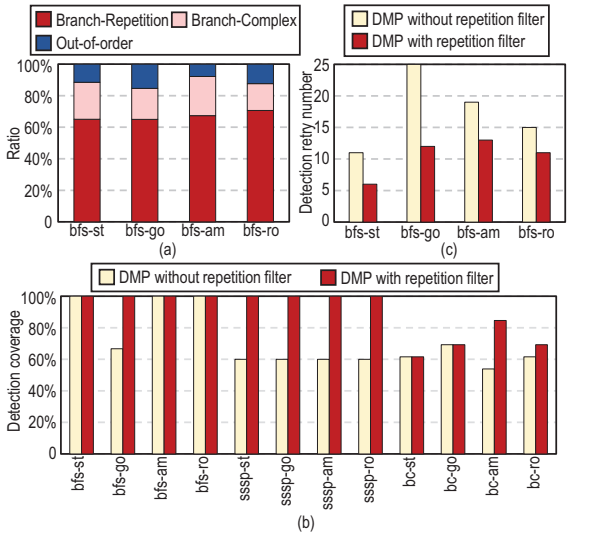


Fig. 16. (a) Breakdown of disturbed memory access into branch-repetition, branch-complex and out-of-order. (b) Indirect pattern detection coverage for DMP with or without repetition filter. (c) Prefetch detection efficiency for DMP with or without repetition filter.

1) Effectiveness of Repetition Filter: Memory access sequence can be disturbed by both branch misprediction and out-of-order execution. To estimate the influence of branch misprediction and out-of-order, Fig. 16(a) classifies the disturbed memory accesses and shows that such incorrect ordering is mostly caused by branch misprediction with only 11.8% from out-of-order execution. While branch misprediction causes not only simple repeated accesses but also more complex patterns, we can observe simple repetition is the dominant factor in Fig. 16(a). Therefore, by using filter to remove repetitive access, DMP effectively mitigates the effect of disordered accesses. To better study its benefits, Fig. 16(b) compares the detection coverage with and without repetition filter. We can observe that for complicated kernels like SSSP and BC, repetition filter effectively helps to detect more indirection patterns. For BFS whose pattern is relatively simple, DMP without repetition filter can achieve acceptable coverage because there are fewer indirect patterns so that DMP can leverage the retry mechanism to tolerate the disordered access sequence. Even in this case, using repetition filter helps DMP faster detect all indirect patterns with much less retry, as shown in Fig. 16(c).

2) Effectiveness of Prefetch Buffer in D-Cache: To evaluate the influence of the position of prefetched data on performance, we manually modify the memory hierarchy of the simulated system to determine the position of prefetched data to L1 D-Cache or L2 Cache. The performance speedups of these two configurations are results are shown in Fig. 17(a). For simple kernels like SpMV, Prefetching data into L1 D-Cache provides a slightly better performance gain(less than 5%). Because data in L1 D-Cache are closer to the CPU pipeline and load latency is lower. But for complicated kernels like BFS, Prefetching data into L2 Cache achieves a significant speedup of 1.47× over prefetching data into D-Cache. Because

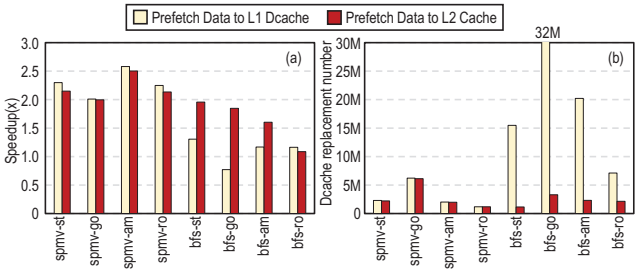


Fig. 17. (a) Execution time speedup over non-prefetching baseline by DMP (prefetch data into L1 Dcache) and DMP (prefetch data into L2 Cache).(b) L1 Dcache replacement number for DMP (prefetch data into L1 Dcache) and DMP (prefetch data into L2 Cache).

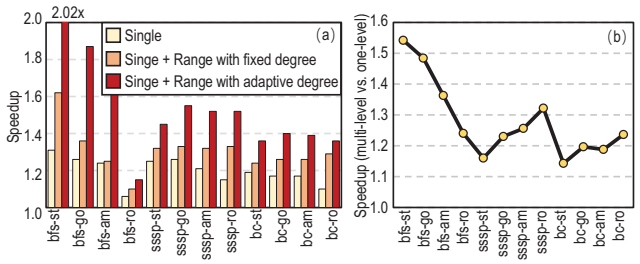
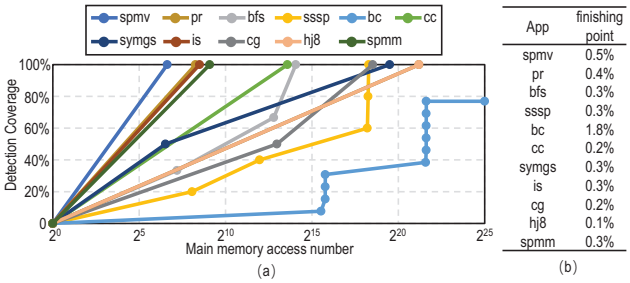


Fig. 18. (a) Comparison among single prefetching, fixed-degree ranged prefetching, and adaptive-degree ranged prefetching. (b) Comparison between one-level and multi-level prefetching.

complicated kernels typically have multiple indirect patterns that need to be prefetched, the cache is easy to be polluted by inserting too many prefetched data and incurs a significant number of cache replacements as Fig. 17(b) shows. Therefore, DMP uses a prefetch buffer to hold prefetched data in D-Cache for prefetching address generation, and the prefetched data are actually placed in the L2 Cache.

3) Benefits of Different Prefetching Types: Fig. 18(a) studies the benefits of ranged prefetching. Compared with single prefetching(16985um^2) that can't prefetch ranged indirect pattern, using ranged prefetching with a fixed degree achieves 1.09× speedup with an additional 4% area cost (17665um^2), while adaptive degree prediction can further improve performance by 1.17× with extra 5% area (18549um^2). Fig. 18(b) shows that compared with one-level prefetching(15811um^2), multi-level prefetching improves 27% performance with only 17% extra hardware overhead (18549um^2). Therefore, considering the considerable performance gain, it is beneficial for DMP to support multi-level and ranged prefetching with acceptable hardware overhead.

4) Detection Efficiency: Fig. 19(a) shows the progress of indirect pattern detection along with the memory access amounts and Fig. 19(b) shows the percentage of execution time when DMP finishes detection. We have two observations. First, DMP can quickly detect the pattern within 1% of the total execution time for most workloads. Second, when indirection patterns get more complicated, DMP needs more time to detect all the patterns. For BC, the most complex workloads in our benchmarks, DMP fails to detect all its indirection patterns as the number of its patterns exceeds the IRT size.



App	finishing point
spmv	0.5%
pr	0.4%
bfs	0.3%
sssp	0.3%
bc	1.8%
cc	0.2%
symgs	0.3%
is	0.3%
cg	0.2%
hj8	0.1%
spmm	0.3%

Fig. 19. (a) The relation between memory access number and the process of indirect pattern detection. (b) Percentage of time when DMP finishes detection for all indirect patterns in workloads.

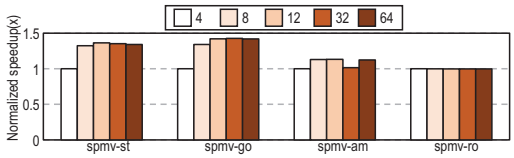


Fig. 20. Design space exploration for PSHR size. The performance of each configuration is normalized to 4 PSHR size.

5) Influence of PSHR Size: We evaluate the speedups of DMP with different PSHR size configurations as shown in Fig. 20. We have two observations from this figure. First, the optimal configuration of PSHR size has a speedup of 1.23× over four PSHR size configurations. Considering the trade-off between performance and storage overhead, we set the PSHR size to 12. Second, the performance gain of adding PSHR size is different for different data inputs. For example, the data distribution of the graph *ro* is relatively regular and has good spatial locality. Therefore, less PSHR size can obtain the optimal performance gain. For graph *am*, 32 PSHR size configuration hurts performance because too many late prefetches are issued and the cache is polluted.

6) Prefetching Effect of Multiple Levels: Fig. 21 shows the effects of multiple prefetching levels for BFS. We can observe that when the level gets deeper, DMP remains high timeliness, but its accuracy and coverage decline because the ranged indirect accesses are difficult to be accurately and fully prefetched. Meanwhile, even at the deepest level, DMP can achieve high accuracy (82%) and coverage (73%).

7) Influence of Sample Window Size: Sample window size is determined to guarantee that the correct PC can be selected as the candidate with high probability. To study the influence, we calculate the average probability of successful selection with different window sizes as shown in Fig. 22. Since the target PC may be overshadowed by other cache misses, enlarging the window size can increase the chance of finding the target PC. As shown in Fig. 22, with window size 64, target-PC can be found in most cases with 85% probability. Therefore, we imperially use 64 window size in our design.

8) Discussion on Scalability: We evaluate the performance of DMP in 1-core to 24-core system with a bandwidth limit of 128GB/s. The results in Fig. 23 show that using 24 cores will saturate the bandwidth and limit prefetching benefits. In this case, DMP reduces the prefetch levels and degree to be less aggressive. Experiments show that this approach achieves

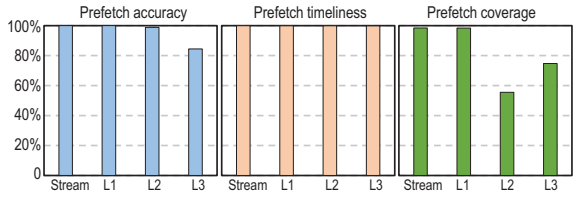


Fig. 21. Prefetch stats for different nested levels on BFS algorithm.

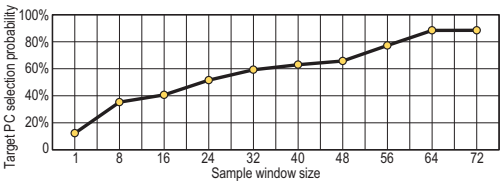


Fig. 22. The probability of the target PC being selected as the candidate for different sizes of sample windows.

13% performance gains for DMP in 24-core system.

E. Hardware Overhead

Fig. 24 shows the average speedup of all kernels in Table IV along with the storage requirements of different prefetchers. Event-Trigger [4] gains 1.5× speedup with a large 8KB storage, while Berti (2.5KB) [57] requires more than 2 times storage space compared to IMP (0.7KB) [75] and IPCP (0.9KB) [59] but only gets a similar performance return. Our design DMP achieves more than 2× performance with an acceptable storage overhead of 0.9KB.

VI. Related Work

Indirect memory access is important in several domains including graph analytics, machine learning, High-Performance Computing (HPC), Warehouse-scale computer (WSC) applications, and general-purpose applications. Many important classes of algorithms including graph algorithms (e.g., GAPS benchmark [11]), machine learning (e.g., sparse CNN [27], graph neural network [41]), HPC (e.g., global dot products, sparse triangular solve [23]), WSC [5] (e.g. web search, knowledge graph) and general-purpose applications (e.g., 12 applications in SPEC CPU2006 [26]) share this kind of memory access pattern. This section discusses the work alleviating the memory access bottleneck for these workloads.

Software prefetching utilizes compilers to perform code analysis to insert prefetching instructions statically [3], [17], [36], [39], [44], [49], [68], [71]. However, such methods may greatly inflate kernel size with prefetching instructions which lowers CPU throughput. Ainsworth proposed a method [68] to automatically insert prefetch instructions for indirect memory accesses However, due to fixed injection sites of prefetching instructions, it is likely to incur too early or too late prefetches.

Software-hardware prefetchers combine static code analysis and dynamic runtime information for more adaptive prefetching [4], [46], [67], [74]. In these methods, irregular patterns are explicitly annotated to configure hardware during run-time. Prodigy [67] programs hardware based on *Data Indirection Graph* from compiler analysis. Such methods can

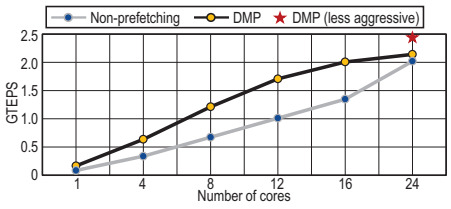


Fig. 23. Average performance of DMP for BFS with different numbers of cores (measured by giga-traversed edges per second (GTEPS)).

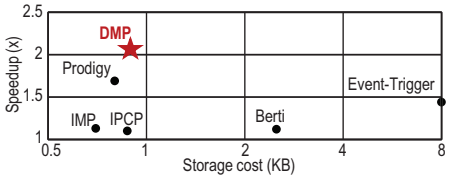


Fig. 24. Storage cost and Execution time speedup for DMP and compared prefetchers. Speedup is normalized to the non-prefetching baseline.

handle complex patterns, but require both software and hardware implementation efforts. They also require kernels to be recompiled, which limits compatibility with legacy workloads.

Compared with the above approaches, **hardware prefetchers** are more difficult to design because it has to detect access patterns at runtime without any software or compiler information. The prediction of hardware prefetchers usually leverages spatial or temporal history. Spatial prefetchers [6], [8], [15], [20], [25], [33], [40], [64] attempt to learn repeating periodic access patterns within a spatial data region. Meanwhile, the temporal prefetchers [9], [13], [21], [28], [34], [37], [47], [70], [72] record the timing order of each access which may require a huge amount of storage for prefetch metadata. However, both spatial and temporal prefetchers only capture limited access patterns, leading to poor behavior when encountering complicated access patterns. Recently, hardware prefetching with more sophisticated detection methods has been proposed for more complicated patterns [14], [18], [38], [60], [65], [75].

Run-ahead [52]–[56] leverage CPU's memory stall time to prefetch useful data. Ajeya proposed Vector Runahead [55] that speculatively reorders scalar operations into vector format to prefetch indirect access. However, unlike DMP, vector runahead requires heavy modification of the core design, which limits its application in commercial CPUs.

Graph accelerators [1], [10], [50], [51], [61], [62] have also been proposed , uses pipelining data [50], [61], intelligent caching [10], [51] and near/in-memory processing [1], [62] to improve memory access. These architectures can integrate our prefetcher to enhance their performance.

VII. Conclusion

This paper proposes DMP, a novel hardware prefetcher to detect indirect memory access patterns using index streams. With the differential matching method, DMP can fast identify an indirect access pattern in pair with its corresponding index stream. Evaluations show that DMP achieves remarkable prefetching efficiency on multiple memory-bound irregular workloads from fields of graphs, databases, and HPC. DMP

alleviates the memory access bottleneck for these workloads and obtains a great performance improvement. DMP improves performance by an average 1.2× speedup against state-of-the-art compiled-based prefetcher. We believe DMP would encourage the next generation of efficient hardware prefetchers targeting more general and complex access patterns.

VIII. Acknowledgment

The authors thank the anonymous reviewers for their insightful comments and suggestions. This work was supported in part by National Key Research and Development Program of China No.2022YFB4500500, National Natural Science Foundation of China No. 62302381 and No.62088102, Key Research and Development Program of Shaanxi No.2022ZDLGY01-08. The authors are with the National Key Laboratory of Human-Machine Hybrid Augmented Intelligence, National Engineering Research Center of Visual Information and Applications, and Institute of Artificial Intelligence and Robotics, Xi'an Jiaotong University, Xi'an, Shaanxi, China.

References

[1] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 2015, pp. 105–117.

[2] S. Ainsworth and T. M. Jones, "Graph prefetching using data structure knowledge," in *Proceedings of the 2016 International Conference on Supercomputing*, ser. ICS '16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2925426.2926254

[3] S. Ainsworth and T. M. Jones, "Software prefetching for indirect memory accesses," in *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2017, pp. 305–317.

[4] S. Ainsworth and T. M. Jones, "An event-triggered programmable prefetcher for irregular workloads," ser. ASPLOS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 578–592. [Online]. Available: https://doi.org/10.1145/3173162.3173189

[5] 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*, 2020, pp. 513–526.

[6] 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.

[7] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, "The nas parallel benchmarks summary and preliminary results," in *Supercomputing '91:Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, 1991, pp. 158–165.

[8] M. Bakhshalipour, M. Shakerinava, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Bingo spatial data prefetcher," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019.

[9] M. Bakhshalipour, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Domino temporal data prefetcher," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 131–142.

[10] A. Basak, S. Li, X. Hu, S. M. Oh, X. Xie, L. Zhao, X. Jiang, and Y. Xie, "Analysis and optimization of the memory hierarchy for graph processing workloads," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2019, pp. 373–386.

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

[12] S. Beamer, K. Asanovic, and D. Patterson, "Locality exists in graph processing: Workload characterization on an ivy bridge server," in *2015 IEEE International Symposium on Workload Characterization*, 2015, pp. 56–65.

[13] 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 (Cat. No.99CB36367)*, 1999, pp. 54–63.

[14] R. Bera, K. Kanellopoulos, A. Nori, T. Shahroodi, S. Subramoney, and O. Mutlu, "Pythia: A customizable hardware prefetching framework using online reinforcement learning," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 1121–1137. [Online]. Available: https://doi.org/10.1145/3466752.3480114

[15] R. Bera, A. V. Nori, O. Mutlu, and S. Subramoney, "Dspatch: Dual spatial pattern prefetcher," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 531–544.

[16] S. Blanas, Y. Li, and J. M. Patel, "Design and evaluation of main memory hash join algorithms for multi-core cpus," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 37–48. [Online]. Available: https://doi.org/10.1145/1989323.1989328

[17] D. Callahan, K. Kennedy, and A. Porterfield, "Software prefetching," in *International Conference on Architectural Support for Programming Languages amp; Operating Systems*, 1991, pp. 40–52.

[18] M. Cavus, R. Sendag, and J. J. Yi, "Informed prefetching for indirect memory accesses," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 17, no. 1, pp. 1–29, 2020.

[19] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "TVM: An automated End-to-End optimizing compiler for deep learning," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 578–594. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/chen

[20] T.-F. Chen and J.-L. Baer, "Effective hardware-based data prefetching for high-performance processors," *IEEE transactions on computers*, vol. 44, no. 5, pp. 609–623, 1995.

[21] Y. Chou, "Low-cost epoch-based correlation prefetching for commercial applications," in *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*. IEEE, 2007, pp. 301–313.

[22] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, pp. 1–25, 2011.

[23] J. Dongarra and M. A. Heroux, "Toward a new metric for ranking high performance computing systems," *Sandia Report, SAND2013-4744*, vol. 312, p. 150, 2013.

[24] B. Falsafi and T. F. Wenisch, *A Primer on Hardware Prefetching*. Morgan & Claypool Publishers, 2014.

[25] A. Fuchs, S. Mannor, U. Weiser, and Y. Etsion, "Loop-aware memory prefetching using code block working sets," in *IEEE*, 2014, pp. 533–544.

[26] J. L. Henning, "Spec cpu2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

[27] T. Hoefler, D. Alistarh, T. Ben-Nun, N. Dryden, and A. Peste, "Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks," *The Journal of Machine Learning Research*, vol. 22, no. 1, pp. 10 882–11 005, 2021.

[28] Z. Hu, M. Martonosi, and S. Kaxiras, "Tcp: Tag correlating prefetchers," in *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings.* IEEE, 2003, pp. 317–326.

[29] Intel Core i3-6006U, https://en.wikichip.org/wiki/intel/core_i3/i3-6006u.

[30] Intel Skylake microarchitecture, https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(client).

[31] Intel Xeon Platinum 8180M, https://en.wikichip.org/wiki/intel/xeon_platinum/8180m.

[32] Intel® 64 and IA-32 Architectures Optimization Reference Manual, https://www.intel.com/content/www/us/en/content-details/671488/intel-64-and-ia-32-architectures-optimization-reference-manual.html.

[33] A. Jain and C. Lin, "Linearizing irregular memory accesses for improved correlated prefetching," in *IEEE/ACM International Symposium on Microarchitecture*, 2013, pp. 247–259.

[34] A. Jain and C. Lin, "Linearizing irregular memory accesses for improved correlated prefetching," in *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2013, pp. 247–259.

[35] A. V. Jamet, L. Alvarez, D. A. Jiménez, and M. Casas, "Characterizing the impact of last-level cache replacement policies on big-data workloads," in *2020 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2020, pp. 134–144.

[36] 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*, 2022, pp. 747–764.

[37] D. Joseph and D. Grunwald, "Prefetching using markov predictors," in *Proceedings of the 24th annual international symposium on Computer architecture*, 1997, pp. 252–263.

[38] A. M. Kaushik, G. Pekhimenko, and H. Patel, "Gretch: A hardware prefetcher for graph analytics," *ACM Trans. Archit. Code Optim.*, vol. 18, no. 2, feb 2021. [Online]. Available: https://doi.org/10.1145/3439803

[39] M. Khan and E. Hagersten, "Resource conscious prefetching for irregular applications in multicores," in *2014 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV)*, 2014, pp. 34–43.

[40] J. Kim, S. H. Pugsley, P. V. Gratz, A. Reddy, and Z. Chishti, "Path confidence based lookahead prefetching," in *IEEE/ACM International Symposium on Microarchitecture*, 2016.

[41] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[42] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe, "The tensor algebra compiler," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, oct 2017. [Online]. Available: https://doi.org/10.1145/3133901

[43] O. Kocberber, B. Grot, J. Picorel, B. Falsafi, K. Lim, and P. Ranganathan, "Meet the walkers: Accelerating index traversals for in-memory databases," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, 2013, pp. 468–479.

[44] M. H. Lipasti, W. J. Schmidt, S. R. Kunkel, and R. R. Roediger, "Spaid: software prefetching in pointer- and call-intensive environments," in *International Symposium on Microarchitecture*, 1995.

[45] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, B. Beckmann, S. Bharadwaj, G. Black, G. Bloom, B. R. Bruce, D. R. Carvalho, J. Castrillon, L. Chen, N. Derumigny, S. Diestelhorst, W. Elsasser, C. Escuin, M. Fariborz, A. Farmahini-Farahani, P. Fotouhi, R. Gambord, J. Gandhi, D. Gope, T. Grass, A. Gutierrez, B. Hanindhito, A. Hansson, S. Haria, A. Harris, T. Hayes, A. Herrera, M. Horsnell, S. A. R. Jafri, R. Jagtap, H. Jang, R. Jeyapaul, T. M. Jones, M. Jung, S. Kannoth, H. Khaleghzadeh, Y. Kodama, T. Krishna, T. Marinelli, C. Menard, A. Mondelli, M. Moreto, T. Mück, O. Naji, K. Nathella, H. Nguyen, N. Nikoleris, L. E. Olson, M. Orr, B. Pham, P. Prieto, T. Reddy, A. Roelke, M. Samani, A. Sandberg, J. Setoain, B. Shingarov, M. D. Sinclair, T. Ta, R. Thakur, G. Travaglini, M. Upton, N. Vaish, I. Vougioukas, W. Wang, Z. Wang, N. Wehn, C. Weis, D. A. Wood, H. Yoon, and Éder F. Zulian, "The gem5 simulator: Version 20.0+," 2020.

[46] Y. Luo, P. Behnam, K. Thorat, Z. Liu, H. Peng, S. Huang, S. Zhou, O. Khan, A. Tumanov, C. Ding, and T. Geng, "Codg-reram: An algorithm-hardware co-design to accelerate semi-structured gnns on reram," in *2022 IEEE 40th International Conference on Computer Design (ICCD)*, 2022, pp. 280–289.

[47] P. Michaud, "Best-offset hardware prefetching," in *IEEE International Symposium on High Performance Computer Architecture*, 2016.

[48] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee, "Measurement and analysis of online social networks," in *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, 2007, pp. 29–42.

[49] T. Mowry and A. Gupta, "Tolerating latency through software-controlled prefetching in shared-memory multiprocessors," *Journal of parallel and Distributed Computing*, vol. 12, no. 2, pp. 87–106, 1991.

[50] A. Mukkara, N. Beckmann, M. Abeydeera, X. Ma, and D. Sanchez, "Exploiting locality in graph analytics through hardware-accelerated traversal scheduling," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 1–14.

[51] A. Mukkara, N. Beckmann, and D. Sanchez, "Phi: Architectural support for synchronization-and bandwidth-efficient commutative scatter updates," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 1009–1022.

[52] O. Mutlu, H. Kim, and Y. Patt, "Address-value delta (avd) prediction: increasing the effectiveness of runahead execution by exploiting regular memory allocation patterns," in *38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)*, 2005, pp. 12 pp.–244.

[53] O. Mutlu, J. Stark, C. Wilkerson, and Y. Patt, "Runahead execution: An effective alternative to large instruction windows," *IEEE Micro*, vol. 23, no. 6, pp. 20–25, 2003.

[54] O. Mutlu, H. Kim, and Y. Patt, "Techniques for efficient processing in runahead execution engines," in *32nd International Symposium on Computer Architecture (ISCA'05)*, 2005, pp. 370–381.

[55] A. Naithani, S. Ainsworth, T. M. Jones, and L. Eeckhout, "Vector runahead," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021, pp. 195–208.

[56] A. Naithani, J. Feliu, A. Adileh, and L. Eeckhout, "Precise runahead execution," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 397–410.

[57] A. Navarro-Torres, B. Panda, J. Alastruey-Benedé, P. Ibáñez, V. Viñals-Yúfera, and A. Ros, "Berti: an accurate local-delta data prefetcher," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022, pp. 975–991.

[58] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bring order to the web," Technical report, stanford University, Tech. Rep., 1998.

[59] S. Pakalapati and B. Panda, "Bouquet of instruction pointers: Instruction pointer classifier-based spatial hardware prefetching," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 118–131.

[60] L. Peled, S. Mannor, U. Weiser, and Y. Etsion, "Semantic locality and context-based prefetching using reinforcement learning," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, 2015.

[61] S. Rahman, N. Abu-Ghazaleh, and R. Gupta, "Graphpulse: An event-driven hardware accelerator for asynchronous graph processing," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 908–921.

[62] S. Rahman, M. Afarin, N. Abu-Ghazaleh, and R. Gupta, "Jetstream: Graph analytics on streaming data with event-driven hardware accelerator," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 1091–1105.

[63] A. Seznec, "A new case for the tage branch predictor," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, 2011, pp. 117–127.

[64] M. Shevgoor, S. Koladiya, C. Wilkerson, Z. Chishti, and R. Balasubramonian, "Efficiently prefetching complex address patterns," in *IEEE/ACM International Symposium on Microarchitecture*, 2017.

[65] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt, "Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers," in *2007 IEEE 13th International Symposium on High Performance Computer Architecture*. IEEE, 2007, pp. 63–74.

[66] Synopsys DC Ultra, https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/dc-ultra.html.

[67] N. Talati, K. May, A. Behroozi, Y. Yang, K. Kaszyk, C. Vasiladiotis, T. Verma, L. Li, B. Nguyen, J. Sun, J. M. Morton, A. Ahmadi, T. Austin, M. O'Boyle, S. Mahlke, T. Mudge, and R. Dreslinski, "Prodigy: Improving the memory latency of data-indirect irregular workloads using hardware-software co-design," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021, pp. 654–667.

[68] S. A. Timothy and M. Jones, "Software prefetching for indirect memory accesses," in *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2017.

[69] TSMC 28nm Technology, https://www.tsmc.com/english/dedicatedFoundry/technology/logic/l_28nm.

[70] T. F. Wenisch, M. Ferdman, A. Ailamaki, B. Falsafi, and A. Moshovos, "Practical off-chip meta-data for temporal memory streaming," in *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*, 2009.

[71] S. Wiel and D. J. Lilja, "A compiler-assisted data prefetch controller," in *International Conference on Computer Design*, 1999.

[72] H. Wu, K. Nathella, J. Pusdesris, D. Sunwoo, and C. Lin, "Temporal prefetching without the off-chip metadata," in *the 52nd Annual IEEE/ACM International Symposium*, 2019.

[73] T. Xia, G. Fu, C. Li, Z. Luo, L. Zhang, R. Chen, W. Zhao, N. Zheng, and P. Ren, "A comprehensive performance model of sparse matrix-vector multiplication to guide kernel optimization," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 2, pp. 519–534, 2023.

[74] M. Yan, X. Hu, S. Li, A. Basak, H. Li, X. Ma, I. Akgun, Y. Feng, P. Gu, L. Deng *et al.*, "Alleviating irregularity in graph analytics acceleration: A hardware/software co-design approach," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 615–628.

[75] 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-48. New York, NY, USA: Association for Computing Machinery, 2015, p. 178–190. [Online]. Available: https://doi.org/10.1145/2830772.2830807

[76] H. Zhao, T. Xia, C. Li, W. Zhao, N. Zheng, and P. Ren, "Exploring better speculation and data locality in sparse matrix-vector multiplication on intel xeon," in *2020 IEEE 38th International Conference on Computer Design (ICCD)*, 2020, pp. 601–609.