2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)

RnR: A Software-Assisted Record-and-Replay Hardware Prefetcher

Chao Zhang
Lehigh University
chz616@lehigh.edu

Yuan Zeng
Lehigh University
yuz615@lehigh.edu

John Shalf
Lawrence Berkeley National Lab
jshalf@lbl.gov

Xiaochen Guo
Lehigh University
xig515@lehigh.edu

***Abstract*—Applications with irregular memory access patterns do not benefit well from the memory hierarchy as applications that have good locality do. Relatively high miss ratio and long memory access latency can cause the processor to stall and degrade system performance. Prefetching can help to hide the miss penalty by predicting which memory addresses will be accessed in the near future and issuing memory requests ahead of the time. However, software prefetchers add instruction overhead, whereas hardware prefetchers cannot efficiently predict irregular memory access sequences with high accuracy. Fortunately, in many important irregular applications (e.g., iterative solvers, graph algorithms, and sparse matrix-vector multiplication), memory access sequences repeat over multiple iterations or program phases. When the patterns are long, a conventional spatial-temporal prefetcher can not achieve high prefetching accuracy, but these repeating patterns can be identified by programmers.**

In this work, we propose a software-assisted hardware prefetcher that focuses on repeating irregular memory access patterns for data structures that cannot benefit from conventional hardware prefetchers. The key idea is to provide a programming interface to *record* cache miss sequence on the first appearance of a memory access pattern and prefetch through *replaying* the pattern on the following repeats. The proposed Record-and-Replay (RnR) prefetcher provides a lightweight software interface so that the programmers can specify in the application code: 1) which data structures have irregular memory accesses, 2) when to start the recording, and 3) when to start the replay (prefetching). This work evaluated three irregular workloads with different inputs. For the evaluated workloads and inputs, the proposed RnR prefetcher can achieve on average 2.16× speedup for graph applications and 2.91× speedup for an iterative solver with a sparse matrix-vector multiplication kernel. By leveraging the knowledge from the programmers, the proposed RnR prefetcher can achieve over 95% prefetching accuracy and miss coverage.

***Keywords*—hardware prefetcher; irregular memory access; iterative algorithm**

I. Introduction

Modern processors have a large processor-memory frequency gap [34]. Small and fast on-chip caches are beneficial to applications that have good spatial and temporal locality. However, many important applications (e.g., web search [14], recommendation system [18], machine learning [56], and scientific computing [40]) have large memory footprints and poor locality. As transistor scaling slows down, caching provides limited benefits for these applications. Prefetchers can help to hide long memory access latency and improve performance by predicting which data will be needed and fetching them earlier from the off-chip memory. Various hardware prefetchers have been proposed to predict future memory accesses based on access sequence observed in the past [9], [10], [25], [36], [38], [41], [47], [52], [53], [60]. Commercial high-performance processors have also adopted increasingly sophisticated hardware prefetchers [21], [30], [51].

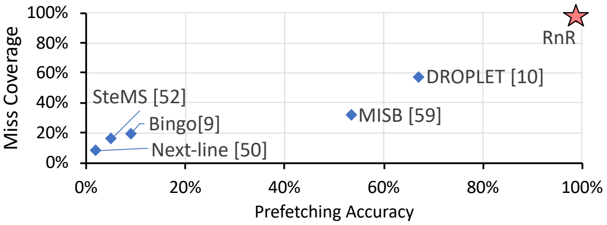
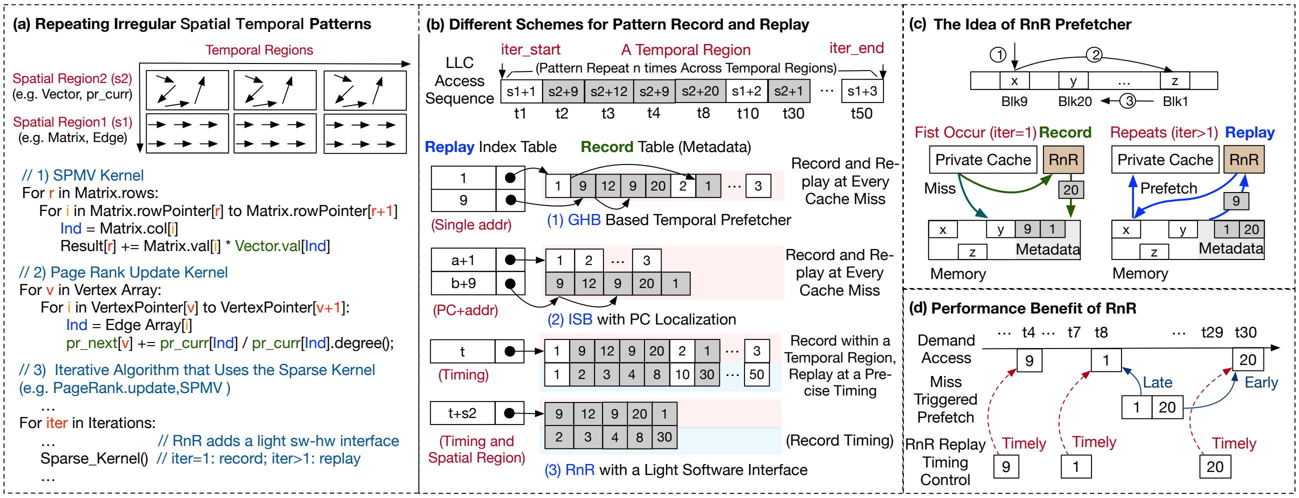


Fig. 1. Prefetcher coverage and accuracy of PageRank [48] on amazon graph [32].

Prefetcher designs are typically tailored to common behaviors in application memory access patterns. Regular memory access patterns like streaming or stride are easy to detect and predict using simple hardware prefetchers [45], [46], [55]. Irregular memory accesses patterns require complicated hardware designs for recording memory access history or training model parameters to predict future access patterns. In most of the hardware prefetchers, common patterns are recorded in shared tables during the program execution. Because these pattern tables have limited capacity, hardware prefetchers face the challenge of differentiating similar memory accessed patterns [9], [25], [36], [38], [41], [47], [52], [53]. In addition, hardware prefetchers do not know precisely when will the data be needed. One class of hardware prefetcher focus on identifying spatial correlations rather than the order or timing of memory accesses [9], [47], [53]. Another class of hardware prefetchers exploit temporal correlations [8], [25], [38], [58], [59]. Meanwhile, commonly used data structures (*e.g.*, graph representations [44], [48] and sparse matrix storage format [23]) can provide information on which data will be needed in what order. Software prefetchers [5], [7] reply on programmer knowledge or common program behaviors to determine how to issue prefetch instructions. However, software prefetchers adds instructions to generate addresses, which might offset the performance gains from prefetching. Moreover, software prefetchers do not know the runtime dynamics of the system—bus congestion and cache contention can cause variable latency for prefetches to return data. Hence, software prefetchers face the challenge of issuing timely prefetches.

An ideal prefetching mechanism should 1) accurately identify when and which data to prefetch and 2) decouple address generation from processors. In this work, a novel software-assisted record-and-replay (RnR) hardware prefetcher is pro-

978-1-7281-7383-2/20/$31.00 ©2020 IEEE
DOI 10.1109/MICRO50266.2020.00057



```
// 1) SPMV Kernel
For r in Matrix.rows:
    For i in Matrix.rowPointer[r] to Matrix.rowPointer[r+1]
        Ind = Matrix.col[i]
        Result[r] += Matrix.val[i] * Vector.val[Ind]

// 2) Page Rank Update Kernel
For v in Vertex Array:
    For i in VertexPointer[v] to VertexPointer[v+1]:
        Ind = Edge Array[i]
        pr_next[v] += pr_curr[Ind] / pr_curr[Ind].degree();

// 3)  Iterative Algorithm that Uses the Sparse Kernel
(e.g. PageRank.update,SPMV )
    ...
For iter in Iterations:
    ...                 // RnR adds a light sw-hw interface
    Sparse_Kernel()  // iter=1: record; iter>1: replay
    ...
```

Fig. 2. Motivation and key idea of the RnR prefetcher.

posed to allow programmers to control when and which data to prefetch. RnR prefetcher provides light programming interface to identify which data structures and code regions in the applications have repeating memory access patterns. The cache misses due to the first appearance of the targeted memory accesses on the predefined data structures will be *recorded* in memory. Timing information of when will the data be used with respect to the execution of the program is recorded in the metadata for timely prefetches in the future. The prefetcher will be invoked to *replay* the recorded address sequence and fetch the corresponding data blocks when repeating memory access patterns are expected during the program execution. The prefetching speed is controlled by hardware to enable timely prefetches. The proposed RnR prefetcher is directed by the software. Therefore, it has better knowledge of which data to record, when to record, and when to replay. The proposed RnR prefetcher hence can achieve close to 100% miss coverage and prefetching accuracy for applications with long repeating irregular memory access patterns. Figure 1 shows the miss coverage and prefetch accuracy comparison of the proposed RnR prefetcher with five existing hardware prefetchers representing different types of prefetchers when running a PageRank application [48]: a regular pattern prefetcher (Next-line [50]), a spatial prefetcher (Bingo [9]), a temporal prefetcher (MISB [59]), a spatial-temporal prefetcher (SteMS [52]), and a domain-specific prefetcher (DROPLET [10]).

The **contributions** of this work are listed below:

(1) This work proposes to separate mixed memory access sequences via programmer-defined temporal and spatial regions. Prior hardware temporal prefetcher designs are trained to recognize common access patterns during the entire execution [38], prior spatial prefetcher designs are trained on memory patterns within a pre-defined spatial region (normally OS page) [53]. These prefetchers differentiate and detect patterns using memory address, offset, program counter (PC), or a combination of them. The proposed work allows programmers to specify static data structures that are accessed repetitively in the same order.

(2) A new prefetch timing control mechanism is proposed. Existing prefetchers issues prefetches triggered by cache access or miss. In this work, the starting point of the replay prefetching is controlled by the software. For the following prefetches, the RnR prefetcher records the timing information as part of the metadata and adjusts the prefetching speed according to the progress of the program execution when issuing replay prefetches.

(3) A software-assisted record-and-replay prefetcher is proposed, which passes information on spatial and temporal regions of interest through a light hardware-software interface to the hardware prefetcher. the proposed RnR prefetcher can achieve a high accuracy and coverage, as well as timeliness, thus can successfully hide the irregular memory access latency and improve system performance.

II. Motivation and Key Idea

The motivation of this work comes from observations on a set of important applications that have repeatitive irregular memory access patterns, which are difficult to predict using conventional hardware prefetchers. Take the Sparse Matrix-Vector multiplication kernel (SpMV) [57] as an example, when the sparse matrix is stored in a CSR format, the row pointer, column array, and matrix value array are accessed in sequence and have good spatial locality. However, the dense vector array is accessed based on the column array of the matrix, which does not have constant stride or other regular patterns (Figure 2 (a)). When the SpMV kernel is used in a sparse conjugate gradient (spCG) solver [20] to get a numerical solution for a partial differential equation, the algorithm starts with an initial guess of the solution and update it according to the conjugate gradients. This process repeats for several iterations until the residual is small enough. This type of iterative solver normally requires tens to hundreds of iterations to converge. In each iteration, the sparse kernel does not change and hence the access sequence to any dense vector that multiplies with the kernel does not change. Similar repeating irregular memory access behaviors are also ubiquitous in iterative graph algorithms (PageRank [39], belief propagation [28], community detection [31], and neighbourhood function approximation [13]).

An observation from the above examples is that the repeating patterns is within certain spatial and temporal regions of the program. As shown in Figure 2 (a), running the spCG or PageRank algorithm produces two kinds of repeating patterns, one type is the regular pattern from reading the matrix array in SpMV, or reading the edge array in PageRank. Another type is the irregular pattern comes from reading the dense vector array in SpMV, or the vertex value (pr_curr). These two different patterns are from two different spatial regions (namely s1 and s2 in Figure 2) of the program since different arrays are stored at different memory location. The two patterns are repeating at the time domain when the program is executing. In this case, an iteration is the temporal region that contains the unique pattern.

When the irregular memory access patterns are long (e.g., large graphs and vectors), existing hardware prefetchers are not efficient to prefetch for these applications. An example is shown in Figure 2 (b). Assume a memory access sequence from two spatial regions s1 and s2 are observed when missed in the last level cache (LLC): one with streaming address pattern 1,2,3, another with irregular address pattern 9,12,9,20,1. These memory accesses arrives at the memory controller within a close time window. This type of behavior is common in SpMV and graph algorithms. For example, SpMV touches both matrix and vector when doing the computation. The mixed pattern could repeat many times when the same sparse matrix is used in an iterative algorithm.

A global history buffer (GHB) [38] based temporal prefetcher will record the global address correlations and use the address as the index to retrieve the pattern. When an address is followed by different addresses in the sequence (*e.g.*, address 9 is followed by 12 and 20), the prefetcher will pick the most recently used one for the prediction and thus may lead to mis-prediction. What's more, a GHB based approach could not differentiate two mixed patterns, thus may lose prefetching opportunity and result in low miss coverage. More recent works such as ISB [25] and MISB [59] utilize PC and address to differentiate different streams and enables efficient metadata management. However, pattern-based prediction faces the fundamental challenge of differentiating similar patterns.

Existing spatial prefetchers [53] also do not work well for this type of applications. The spatial region set for the spatial prefetcher to detect pattern is normally an OS page. The page size is small and no ordering information is recorded within a page. Spatial prefetchers are efficient in observing the same spatial patterns among different spatial regions. In a prior spatial-temporal prefetcher [52], the access order information are recorded among different sequences, but order information within spatial region is not recorded. In the example in Figure 2 (a), the repeating patterns are within a spatial region, but across different temporal region. The size of the spatial region could vary for different data structures and could be much larger than a page size.

The challenges faced by these existing prefetchers can be solved by providing programmer knowledge—if the prefetcher can know when and where will the repeating pattern start and end, it could achieve higher prediction accuracy and prefetch in a timely manner. In this example, if the timing information of when the iteration starts and ends is known by the prefetcher, it could record the cache miss pattern at the first appearance of the sequence, and replay the exactly same sequence for the following repeats. What's more, if the information of the spatial region could be known by the prefetcher (*i.e.,* the address range of the matrix and vector array in this example), the prefetcher could differentiate the streaming pattern from the irregular pattern and thus avoid recording the stream one.

Another observation is that the existing prefetchers could suffer a lot from ill-timed prefetches. An example is shown in Figure 2 (d). Most of the existing hardware prefetchers trigger the prefetches when a demand miss or access happens. Normally, when a demand miss occurs, the prefetcher will prefetch several recorded address in sequence. However, the actual timing information for two correlated address are unknown, which could lead to either early or late prefetches. Prior prefetchers only record miss order, if the timing information corresponding to recorded memory accesses can be stored as part of the metadata as well, more timely prefetches could be issued.

Fortunately, the repeating irregular patterns are not difficult to identify in the source code of applications like graph algorithms and iterative solvers with sparse matrix. First, the spatial regions can be identified by knowing which data structures are accessed in a repeating irregular fashion. The temporal regions can be specified by programmers at the start and end of each iterations or evocations. More precise timing information can be estimated by observing miss ratios with respect to progressing time windows during the first occurrence of a target sequence. Besides performance considerations, metadata management is an important issue for prefetcher design, especially for temporal prefetchers that records data correlations. With software assists, metadata could be stored in a contiguous storage region, which makes metadata prefetching fast and efficient. What's more, since the hardware now equipped with the programmer knowledge of the temporal region, the metadata storage space could be released as soon as the target executing phase ends, other than waiting for the entire program to finish. This scheme provides more efficient metadata management as compared to prior works.

This work proposes a software-assisted record-and-replay prefetcher. The **key ideas** of this design include: 1) leveraging the software knowledge of when the repeating memory access sequence starts and ends to record the cache misses and prefetch by replay; (2) allowing the software to set regions of interest to reduce recording overhead and focus on irregular patterns; and (3) recording timing information as part of the metadata to enable a more timely prefetching.

III. Design Choices

While the key ideas are simple, to design an efficient RnR prefetcher many design choices need to be carefully reviewed. RnR targets accurately recording and replay of repeating irregular sequences using software assisted hardware. In this design, a lightweight software interface is provided to allow the software/programmer to communicate to the hardware the temporal/spatial information about when to record, when to replay, and when to stop recording the memory access sequence. A more detailed explanation of the software-hardware interface will be introduced in Section IV. The next question is what to record. A naive approach is to record every data access within the defined range, which is inefficient. Although the sparse structures have poor spatial and temporal locality in general, sometimes locality does exist, such as neighboring vertexes sharing some common neighbors in a social network graph. Recording all of the structure accesses may lead to redundant record and prefetch, and thus waste the storage space and bandwidth. Therefore, the proposed design chooses to record the miss sequence from the private L2 cache in a multi-level cache hierarchy. At this stage, memory accesses that could benefit from locality are filtered by the private L1 and L2 caches. Regarding where to store the recorded information, the proposed design puts these data into a predefined memory space allocated by the programmer using a programming interface that will be introduced in Section IV. Since the access latency to get the recorded metadata is long, RnR uses a metadata prefetching scheme similar to [25]. Another important question is when to prefetch. Previous designs trigger the prefetch event based on a single cache access, or a single cache miss, or a combination of them with a confidence threshold. RnR starts prefetching at the replay phase. Hardware gets knowledge about the optimal timing and phasing of the prefetch stream through the software interface. Within the replay phase, RnR uses an efficient reply timing-control mechanism to ensure timely prefetching. This is achieved by recording timing information as part of the metadata, which will be introduced in details in Section V-C. Finally, regarding where to put the prefetched data, RnR chooses the private L2 cache. This is based on the observation in DROPLET [10] that using the L2 cache for this purpose shows negligible influence of cache pollution.

IV. Programming Interface

This section describes the architectural states and programming interface of the RnR prefetcher (IV-A), an example use case (IV-B), and the operating system supports (IV-C).

A. Architectural States and RnR Functions

To achieve an effective "Record and Replay" of the memory accesses, the proposed RnR prefetcher requires the following additional architectural states: (1) a address space identifier (ASID) register for permission check, (2) a set of boundary checking address registers [1] and their associated size and active states (enable vs. disable), (3) a base address register for a *Sequence Table*, (4) a *window size* register, (5) a base address register for a window *Division Table*, and (6) a *prefetch state* register. The sequence table is used for recording miss sequence. The window size and window division table are used for recording miss ratios during each time window such that this information can be used to control the replay prefetching pace (Section V-C)). Both the sequence table and the window division table are stored in memory spaces allocated by the programmer, hence the registers only need to store the base addresses of these two tables. All of the these architectural states are implemented as special registers per core. All of the address registers store virtual addresses. The ASID register stores the identifier of the process that is currently using the prefetcher.

The programmer needs to first identify which target data structures to be recorded. The target data structures can be defined at the memory allocation time. By passing down the base addresses and the corresponding sizes to the boundary registers, the prefetcher can recognize whether the access is within the target range or not (Section V-A). Accesses out of the target range will not be recorded or replayed (prefetched).

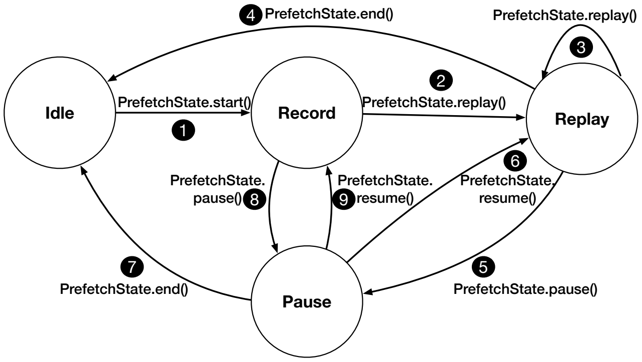


Fig. 3. State transition diagram of the RnR prefetch state.

TABLE I
RnR function calls.

Function	Explanation
RnR.init()	Set ASID for permission check, allocate memory for SequenceTable and DivisionTable, and set the default window size
AddrBase.set(*addr*, *size*)	Add a base address with its corresponding size
AddrBase.enable(*addr*)	Enable the address boundry check for *addr*
AddrBase.disable(*addr*)	Disable the address boundry check for *addr*
WindowSize.set(*size*)	Set a window size different from the default
PrefetchState.start()	Enable RnR, start recording
PrefetchState.replay()	Start replay from the beginning
PrefetchState.end()	Disable RnR
PrefetchState.pause()	Pause replaying
PrefetchState.resume()	Resume replaying from the pause state
RnR.end()	Free the memory space for metadata

Moreover, the software also needs to define where will the miss sequence be stored, 2) how large will be the window size for pace control, and 3) where will the window division table be stored. Window size determines the number of structure misses that are recorded in each of the window. To provide the timing information that indicate how long each of the window is, the window division table stores the total number of demand accesses within each of the window, which is used for controlling prefetching aggressiveness (Section V-C).

The prefetch state is also controlled by the software. ❶ At first, the programmer needs to enable the RnR prefetcher to *start* recording the miss sequence during the first appearance of the sequence. At the end of the recording, a *replay* function can be invoked to disabled the recording and start the replaying

[1] The number of address registers can be variable, two are used in the evaluation.

Algorithm 1: PageRank using RnR Prefetcher.

```
1  Procedure Init():
2      $p_{\text{curr}} = \{\frac{1}{|V|}, ..., \frac{1}{|V|}\}$
3      $p_{\text{next}} = \{0.0, ..., 0.0\}$
4      diff = {}
5      Frontier = $\{0, ..., |V| - 1\}$
6      error = $\infty$
7      RnR.init()
8      RnR.AddrBase.set($p_{\text{curr}}$, N)
9      RnR.AddrBase.set($p_{\text{next}}$, N)
10 return 1
11
12 Procedure PRUpdate(s, d):
13     AtomicIncrement($p_{\text{next}}[d]$, $\frac{p_{\text{curr}}[s]}{deg+[s]}$)
14 return 1
15
16 Procedure PRNormlize(i):
17     $p_{\text{next}}[i] = (\gamma \times p_{\text{next}}[i] + \frac{1-\gamma}{|V|})$
18     diff$[i] = |p_{\text{next}}[i] - p_{\text{curr}}[i]|$
19     $p_{\text{curr}}[i] = 0.0$
20 return 1
21
22 Procedure PageRank($G, \gamma, \epsilon$):
23     Init()
24     RnR.AddrBase.enable($p_{\text{curr}}$)
25     RnR.PrefetchState.start()
26     while ($error > \epsilon$) do
27         EdgeMap(G,Frontier,PRUpdate)
28         VertexMap($\gamma$,Frontier,PRNormlize)
29         error = sum of diff entries
30         SWAP($p_{\text{curr}}, p_{\text{next}}$)
31         RnR.PrefetchState.replay()
32         RnR.AddrBase.enable($p_{\text{next}}$)
33         RnR.AddrBase.disable($p_{\text{curr}}$)
34     end while
35     RnR.PrefetchState.end()
36     RnR.end()
37 return $p_{curr}$
```

❷ (start prefetching from the beginning of the stored sequence in the memory). If the repeating sequence is not expected immediately, a *pause* can be used to stop recording ❽. At the end of each of the replay, either a *replay* ❸ or *pause* ❺ can be used to stop the replay. After all of the repeated iteration finished, and *end* function can be called ❹ to terminate the "Record and Replay" process. The record or replay can be paused ❺ ❽ in the middle and resumed ❻ ❾ from where it is paused to support context switch.

B. An Example of Using the Programming Interface

The PageRank from Ligra [48] is used as an example to show an use case of the programming interface (Algorithm 1, RnR related codes are colored in blue). Note that this example includes an out-of-place update for p_{curr} and p_{next}, which means the base pointer will be exchanged at the end of each iteration (line 33). Therefore the full address range of p_{curr} in each iteration is not the same. However, the element in the array can be divided into *Base+Offset*. Even if the *Base* changes, the *Offset* are kept the same. To correctly select which *Base* to use, the programmer also needs to swap the *Base* at the end of each iteration. Line 7 initializes the RnR prefetcher by setting internal parameters and allocating memory space for metadata. RnR.init() also sets the *Window Size* for replay timing controller (described in Section V-C) to a default value. Since the prefetching destination is the L2 cache as described in Section III, the default window size is set to half of the L2 size for double buffering.

Line 8-9 defines the virtual address range for p_{curr} and p_{next}, with their corresponding size N (N is the number of vertices in this case). Line 24 enables the address range for p_{curr} before the recording starts. Line 25 starts the recording process. Line 31-33 enable and disable the swapped base address p_{curr} and p_{next} for prefetching the next iteration. Line 35 terminates RnR prefetcher. Line 36 free out the reserved storage for all of the metadata stored in memory.

C. Operating System Supports

The operating system can schedule a process or thread to different cores to optimize for resource utilization. A process can also be switched out due to long latency events such as page faults. Conventional hardware prefetchers need retraining after context switching. However, RnR does not need retraining if the same access order is expected because the metadata is stored in allocated heap space. During context switch or process migration [22], the on-going recording or replaying need to be paused. The corresponding architectural states as well as RnR internal states (Section V) need to be copied to memory. For migration and switching in a process that has paused RnR prefetching, the values of these registers will be copied in before resuming the execution of the process and the prefetching. RnR requires to save/restore a total of 86.5B of additional states (Section V) , which is not expected to add a significant performance overhead. This is because context switching overhead comes from losing register, TLB, branch predictor, and cache states. Cache warmup penalties are typically the bottleneck [26], [33].

V. RnR Architecture

Since the programmer can now define when to start and end the record and replay using the programming interface, the proposed RnR prefetcher executes according to the architectural state diagram in Figure 3. In Figure 4, arrow A, B, C, D, and E are software interfaces that updates RnR architectural states to guide the RnR prefetcher. In addition to the registers that are visible to software, the RnR prefetcher also requires internal registers, which include: (1) a current structure read counter; (2) a 128B buffer for sequence table and a 128B buffer for division table; (3) length registers for the two tables respectively; (4) current physical addresses to write or read the two tables; (5) a number of prefetcher counters; (6) a current window counter; and (7) a current prefetch pace register that stores the desired number of demand reads per prefetch during the current prefetch window.

A. Record

After the prefetch state register is set to 'Record', RnR starts to record the L2 miss sequence of the target data structure. Figure 4 shows how does the record iteration performed before the programmer stops recording. ❶ All demand accesses need to check the address boundaries stored in the boundary table before the virtual to physical address translation. ❷ If the

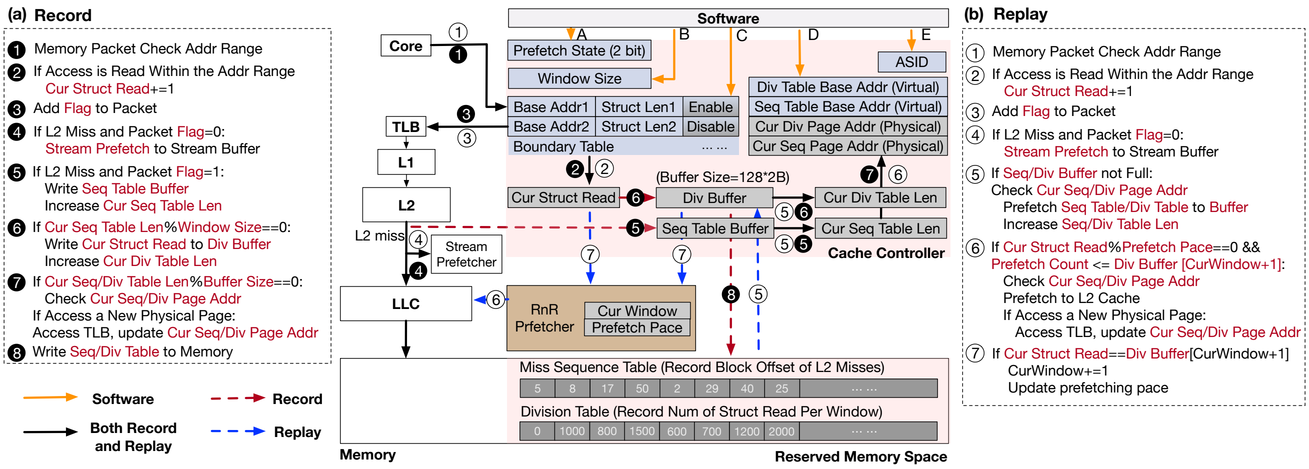

(a) Record
❶ Memory Packet Check Addr Range
❷ If Access is Read Within the Addr Range
Cur Struct Read+=1
❸ Add Flag to Packet
❹ If L2 Miss and Packet Flag=0:
Stream Prefetch to Stream Buffer
❺ If L2 Miss and Packet Flag=1:
Write Seq Table Buffer
Increase Cur Seq Table Len
❻ If Cur Seq Table Len%Window Size==0:
Write Cur Struct Read to Div Buffer
Increase Cur Div Table Len
❼ If Cur Seq/Div Table Len%Buffer Size==0:
Check Cur Seq/Div Page Addr
If Access a New Physical Page:
Access TLB, update Cur Seq/Div Page Addr
❽ Write Seq/Div Table to Memory
Software
Record
Both Record and Replay
Replay
Core
① ❶
TLB
❸ ③
L1
L2
L2 miss ④ ❹
Stream Prefetcher
LLC
⑥
Software
A B C D E
Prefetch State (2 bit)
Window Size
Base Addr1
Struct Len1
Enable
Base Addr2
Struct Len2
Disable
Boundary Table
... ...
ASID
Div Table Base Addr (Virtual)
Seq Table Base Addr (Virtual)
Cur Div Page Addr (Physical)
Cur Seq Page Addr (Physical)
❷ ②
(Buffer Size=128*2B)
Cur Struct Read
❻
Div Buffer
Cur Div Table Len
❼ ⑥
⑤ ❻
❺
Seq Table Buffer
Cur Seq Table Len
⑤ ❺
Cache Controller
⑦ ⑦
❽ ⑤
RnR Prfetcher
Cur Window
Prefetch Pace
Miss Sequence Table (Record Block Offset of L2 Misses)
5 8 17 50 2 29 40 25 ... ...
Division Table (Record Num of Struct Read Per Window)
0 1000 800 1500 600 700 1200 2000 ... ...
Memory
Reserved Memory Space
(b) Replay
① Memory Packet Check Addr Range
② If Access is Read Within the Addr Range
Cur Struct Read+=1
③ Add Flag to Packet
④ If L2 Miss and Packet Flag=0:
Stream Prefetch to Stream Buffer
⑤ If Seq/Div Buffer not Full:
Check Cur Seq/Div Page Addr
Prefetch Seq Table/Div Table to Buffer
Increase Seq/Div Table Len
⑥ If Cur Struct Read%Prefetch Pace==0 &&
Prefetch Count <= Div Buffer [CurWindow+1]:
Check Cur Seq/Div Page Addr
Prefetch to L2 Cache
If Access a New Physical Page:
Access TLB, update Cur Seq/Div Page Addr
⑦ If Cur Struct Read==Div Buffer[CurWindow+1]
CurWindow+=1
Update prefetching pace

Fig. 4. RnR system architecture.

demand access is within the range of an entry in boundary table, the *Cur Struct Read* counter increments. This counter is used for providing the pacing information for the replay. ❸ Demand accesses check the TLB and caches as usual. Only the memory request within the address range will be marked with a flag to let the RnR prefetcher know it is from the targeted data structure in case it is also a L2 miss. ❹ Cache misses out of the RnR prefetch range can be trained by other prefetchers (Section V-D). ❺ If the demand acccess misses in the private L2 cache and within the target range, the RnR prefetcher will write a new entry in the sequence table buffer and increase the sequence table length. ❻ For every X number of misses the sequence table record, where X is the window size, the current structure read count will be recorded in window division table. This provides the miss ratio information of this window, which can be used to guide the prefetch speed during the replay (Section V-C). ❼ To minimize the write traffic, the metadata writes are grouped at cache line granularity (write back every 64B). The RnR prefetcher now needs to find the reserved memory space for writing metadata. The virtual address of the metadata head pointer can be calculated by calculating *Table Base Addr* + *Cur Table Len*. To find the physical address, a TLB lookup is needed. Performing TLB lookups for each metadata write might block demand accesses. Since metadata writes are sequential and have good spatial locality, a current physical page address buffer is added to only perform one TLB lookup per 4MB page. ❽ Finally, the content in the sequence table and window division table are written back to the corresponding addresses. The RnR terminates the entire recording iteration when the programmer changes the prefetch state to 'Pause' or 'Replay'.

B. Replay

When the RnR prefetch state is transitioned to 'Replay', the *Cur Struct Read*, *Cur Div Table Len*, *Cur Seq Table Len* are reset to zero. Every demand access goes through ①,②,③, and ④ is similar to when the prefetch is in a 'Record' state. The *Cur Struct Read* counter will count accesses to the targeted structures again to estimate the progress of the program. ⑤ The miss sequence and timing information inside the sequence table and window division table are proactively prefetched into their corresponding buffers (the 128B buffer size allows double buffering) and the table length is updated. Since the metadata is stored in a contiguous address space, metadata prefetching for the sequence table and the division table has a streaming pattern and does not incur timely lookup. ⑥ Similar to the 'Record' state, the RnR prefetcher needs to calculate the addresses in order to prefetch for both the sequence table and the division table. One TLB lookup is needed for a 4MB page to perform the virtual to physical address translation. Once the address is generated, the prefetch requests could be issued to LLC based on the prefetch pace. If the request hit in the LLC, the data will be fetched to the L2 cache, otherwise the request goes to the memory. ⑦ The prefetching aggressiveness (how much ahead to prefetch) needs to be adjusted to match with the application progress, which can be tuned at the granularity of the prefetching window. The proposed design stores the prefetched blocks into L2 cache, hence the aggressiveness is bounded by the L2 cache size. For the evaluated benchmarks, double buffering (one window ahead) is enough to provide timely prefetches. The RnR prefetcher also adjust the frequency of issuing prefetches within a prefetch window to minimize contention with demand accesses without delaying the prefetches, which will be discussed in Section V-C. When the *Cur Struct Read* matches with the recorded count for the next window, the current window counter is incremented and the RnR can start to prefetch misses for the next window after finishing all of the prefetches for the current window. The prefetching pace will be hence updated with window switching.

C. Replay Timing Control

Traditional memory prefetchers can be triggered for different levels of aggressiveness. Nexline prefetch requests the next cache line after each access. Spatial and temporal prefetchers can trigger an new prefetch request when the miss sequence matches with the recorded common pattern. RnR stores one sequence per core to prefetch. As described in Section V-A, RnR also stores the miss ratio information per prefetch window. This allows the RnR prefetcher to match prefetching speed with the progress of the program execution. As shown in the Figure 5 (a), different window may have different miss

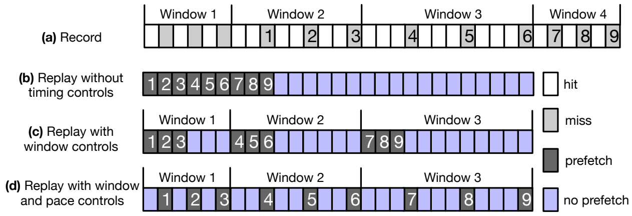


Fig. 5. Prefetching timing control example with window size = 3.

ratios according to whether the data region has relatively good locality or not. In this example, Window 1 has a miss ratio of 50%, whereas the window 2 has a miss ratio of 33.3%. If the prefetcher does not consider the miss ratio information and issues a prefetch on every demand access as shown in Figure 5 (b), the prefetches would be too aggressive and the prefetched data would be easily evicted out of the L2 cache by the time the data is needed. However, if the miss ratio information is stored, the replay prefetch controller can know when to stop. In the example in Figure 5 (c), after the third prefetch is issued, the RnR prefetcher will wait until the sixth access to the targeted data structure before it issues prefetches of the next window. To further optimize the prefetch traffic, the prefetches can be evenly distributed within a prefetch window to avoid congestion and interference with demand accesses. The frequency of issuing prefetch requests can be calculated by $N_{\text{Pace}} = \frac{\text{StructAccessesInCurrentWindow}}{\text{WindowSize}}$. This means a prefetch is issued every N_{Pace} structure accesses (Figure 5 (d)).

D. *Integrate with Other Prefetchers*

As shown in Figure 4, the L2 stream prefetcher excludes the target sparse data structure so that the conventional stream prefetcher [21], [30], [51] will be trained by L2 misses outside of the Record-and-Replay address range. The proposed RnR prefetcher can support both regular and irregular memory accesses, but a streaming or constant stride prefetcher would be more efficient at predicting regular memory access patterns. The proposed RnR prefetcher can be easily integrated with other hardware prefetchers by filtering out temporal and spatial regions of interest.

E. *Scalability for Multicore Systems*

There are two questions to answer regarding the scalability of this design in a multicore system: 1) how will the hardware overhead grow as the number of cores increases? and 2) how will the metadata storage overhead grow as the number of cores increases? RnR collects the private L2 miss sequences from the target data structure. The special registers to store architectural states and RnR internal states are per core. Therefore, the hardware overhead increases linearly with respect to the core count. Given the fact that the total hardware overhead for each core is small, the hardware area overhead is negligible as compared to the chip area (Section VII-B). The total storage overhead depends on the number of L2 cache misses from the target data structure. For parallel workloads, the data structures of a large problem are typically partitioned and assigned to multiple threads to run on different cores. Accesses to another core's data partition might increase the total number of cache accesses. But when more cores are opted in, the total capacity of private caches is also increased. Hence the total L2 misses might be reduced. Moreover, partitioning algorithms [29], [49] typically aim to improve locality and reduce cross-partition communication. After partitioning the data structure, each worker thread executes on different cores will primarily access their own data partition. The data regions of interest should also be set for each data partition. In such cases, the total storage overhead will not increase significantly when running on multicore systems.

VI. Experimental Setup

This work uses ChampSim [3], a trace-based simulation infrastructure, to evaluate the proposed RnR prefetcher. ChampSim is adopted in the 3rd Data Prefetching Championship (DPC3) [1]. The cache subsystem in ChampSim includes FIFO read and prefetch queues. The demand requests have a higher priority than prefetch and metadata requests. The main memory model simulates data bus contention, bank contention, and bus turnaround delays. When there are more contentions, memory access latency is increased. The baseline configuration is shown in Table II. Cache sizes and latency are modeled based on Intel i7-6700 [4]. Cadence Genus Synthesis Solution [15] is used to analyze the RnR prefetcher hardware area. The area is estimated based on FreePDK45nm [54] standard cell library, and is scaled to 22 nm. The memory timing constraint comes from Micron MT40A2G4 DDR4-2400-CL17 data sheet [2]. We mark the region of interest (ROI) of the code and using the PIN tool [35] to extract the kernel trace for trace-based simulation. 20 million instructions are used to warm up the on-chip caches and at least 500 million instructions of the kernel are simulated. [2]

TABLE II
Baseline configuration.

Processors	4 cores, 4 Ghz, 4-wide OoO, 256-entry ROB, 64-entry LSQ perceptron branch predictor [27], 16-entry issue queue
L1-Ds/Is	private, 64KB, 8-way, 8-entry MSHR, delay = 4 cycles
L2s	private, 256KB, 8-way, 16-entry MSHR, delay = 12 cycles
LLC	shared, 8 MB, 16-way, 128-entry MSHR, delay = 42 cycles
Memory Controller	FCFS, read queue size = 64, write queue size = 32 write queue draining: high/low threshold = 75%/25%
Main Memory	4GB, 2400 Mhz, 1 channel, 1 rank, 16 banks tCL = tRCD = tRP = 17 cycles

This work evaluated vertex-centric PageRank algorithm from Ligra [48], sparse CG from Adept [23], and edge-centric Hyper-anf from x-stream [44], which have repeated access trace across different iterations. Graph and sparse matrix inputs of the application are listed in Table III, which covers representative inputs with different characteristics (*i.e.,* size, sparsity). Graph algorithm and scientific applications have many different libraries support for parallelization [16], [24], [43]. In this work, we implement the applications as Single Program Multiple Data (SPMD) [17] model (every task executes the same program), which is commonly used in many pull-based graph algorithm [12] with graph partitioning

[2] Trace sizes for different applications and inputs varies a lot. We simulate the whole record iteration and replay iteration regardless of how large the traces are.

[29], [49]. The master process initializes the array, sends the partitioning information to worker processes, performs its own share of the computation and wait for all of the worker processes to finish. Each worker process receives the partitioning information, performs its share of computation, and sends results back to the master. We use METIS [29] to partition graph inputs into four partitions and assign each of the worker processes a partition to run.

TABLE III
INPUT DATASETS.

Graph Input				
Names	Edges	Nodes	Size	Type
amazon [32]	3.3M	0.4M	47.9MB	purchase network
com-orkut [32]	117M	3.1M	1.7GB	social network
urand [11]	260M	16.8M	2.1GB	synthetic
roadUSA [11]	57.7M	23.9M	1.3GB	road network
Sparse Matrix				
Names	Rows	Nonzeros	Size	Type
atmosmodj [19]	1.2M	8.8M	214MB	fluid dynamics
bbmat [19]	38.7K	1.7M	42.4MB	fluid dynamics
nlpkkt80 [19]	1.1M	28.1M	350MB	PDE optimization
pdb1HYS [19]	36.4K	4.3M	86.0MB	protein data bank

VII. Evaluation Results

The proposed RnR prefetcher is compared against three general-purpose prefetchers and one domain-specific prefetcher for graph algorithm: 1) Next-Line, 2) SteMS [52], 3) BINGO [9], and 4) DROPLET [10]. Since DROPLET is designed for graph algorithms, the evaluation results do not include DROPLET when running spCG.

A. Performance

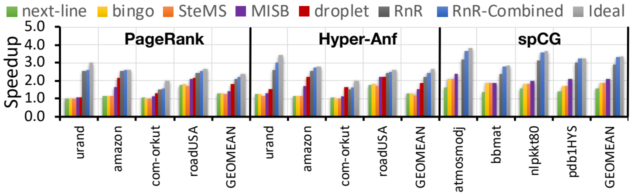


Fig. 6. Speedup over no prefetcher baseline.

1) Speedup: The proposed RnR prefetcher does not prefetch for the target structure during the recording state. The effective performance of RnR depends on how many times it can reuse the recorded pattern to prefetch. The more replays, the higher speedup the RnR can achieve. For PageRank and HyperAnf, the total number of the iterations depends on the graph inputs and the stopping criteria (*e.g.*, Line 29 in Algorithm 1). For all given inputs and the default ϵ value of the benchmark suite, both PageRank and Hyper-Anf take more than a hundred iterations to converge. For spCG, the algorithm converges when the residual is equal or smaller than the default value, which typically takes hundreds of iteration to finish. Therefore, the overhead of the record iteration can easily be amortized by the replay iterations. We use 100 iterations for all tested applications and inputs for simplicity. As shown in Figure 6, RnR refers to using the proposed prefetcher for only the irregularly-accessed data structure, RnR-Combined refers to using the RnR prefetcher for the irregularly-accessed data structure and using stream prefetcher (next-line) for all other data. The ideal case is analyzed by having an infinite-sized LLC.

For graph algorithms, RnR achieves the highest performance improvement for most of the graph inputs and can increase performance significantly for the synthetic graph *urand*, which has random connections and poor data locality. General-purpose prefetchers (next-line, bingo, SteMS) are typically ineffective for the graphs that have many random connections due to the lack of a common spatial or temporal pattern to exploit. DROPLET achieves near zero performance improvement for PageRank with the *urand* graph input. This is because DROPLET first prefetches the edge data in a streaming fashion and then uses the prefetched edge data to generate vertex addresses to prefetch for indirect accesses. The address generation latency of this extra level of indirection can take away the potential opportunity of timely prefetching for random graph like *urand*, in which the vertices are lack of spatial locality and hence take longer time to fetch.

The proposed RnR prefetcher does not have dependencies when generating prefetch addresses. The metadata is prefetched from recording without recomputing. Therefore, the RnR prefetcher is effective for all of the graph inputs and can achieve the highest performance. RoadUSA is a road map and presents a relatively regular connection pattern, so that even the general-purpose prefetchers are effective.

For spCG, RnR prefetcher can deliver higher speedups because it can separate stream and sparse access patterns. In general, the proposed RnR can achieve 2.11×, 2.23×, and 2.90× performance improvement for PageRank, Hyper-Anf, and spCG kernel respectively.

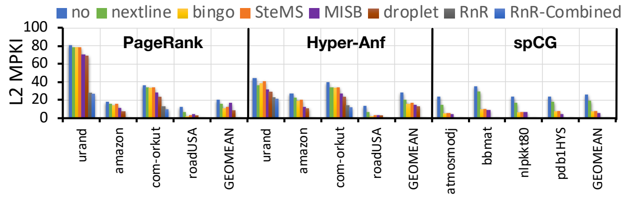


Fig. 7. L2 MPKI.

The proposed RnR prefetcher uses the private L2 as the prefetching destination. To create a fair comparison, all of the evaluated prefetchers are prefetching data into the private L2. We believe this does not handicap the other prefetchers as the partitioned graphs utilized for the parallel benchmarks have minimal sharing between threads. As shown in Figure 7, the proposed RnR-Combined prefetcher can reduce the demand miss ratio by 97.3%, 94.6%, and 98.9% for PageRank, Hyper-Anf, and spCG kernel respectively. And for graph algorithms with urand and com-orkut as the inputs, the MPKI can still be reduced by more than a half.

2) Coverage: The miss coverage refers to the total number of misses that the prefetcher can reduce from the baseline, which is a direct indication of the effectiveness of the overall prefetcher design. 100% coverage means the prefetcher can correctly prefetch all of the misses observed in the baseline (*i.e.*, a "perfect" prefetcher). Hence, *Coverage* = $\frac{\text{Useful Prefetches}}{\text{Total Baseline Misses}}$. The higher the coverage, the more misses the

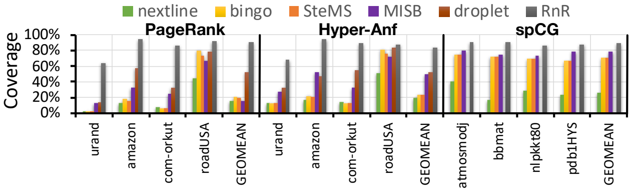


Fig. 8. Miss coverage.

prefetcher can detect and correctly prefetch. Spatial-temporal prefetchers reply on the shared common access pattern, which are not as prevalent in the evaluated applications as database and transaction applications. Therefore, only part of the miss sequence can be detected. For roadUSA with good locality, the bingo and SteMS can achieve good performance. DROPLET covers the dependent load by using edge data to predict the dependent load of the vertex data, which can achieve better coverage for edge-centric Hyper-Anf as compared to vertex-centric PageRank. The proposed RnR prefetcher can detect all of miss sequences for the target data structure without sharing patterns and data dependencies. Therefore, RnR can achieve on an average of 91.4%, 84.5%, and 88.7% of miss coverage.

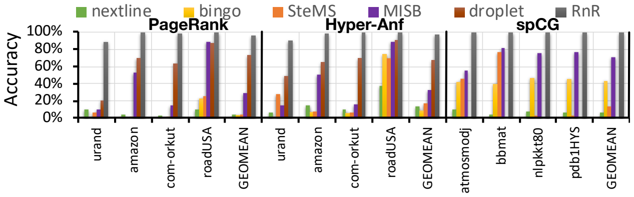


Fig. 9. Prefetcher accuracy.

3) Accuracy: The prefetching accuracy is the fraction of useful prefetches out of the total issued prefetches. *Accuracy* = $\frac{\text{Useful Prefetches}}{\text{Total Prefetches}}$. General purpose prefetchers (bingo and SteMS) have problems with sequence miss-matches for accesses with similar spatial-temporal patterns. Therefore, for applications dominated by irregular memory accesses, general-purpose prefetchers achieve the lowest accuracy. For some inputs with a good spatial locality, for example, roadUSA, they can average nearly 50% accuracy. DROPLET achieves better accuracy as compared to bingo and SteMS. This is because DROPLET is able to capture the data-dependent indirect access for graph algorithms. However, for graph algorithms where the edge array can not be fetched early enough (PageRank-urand), DROPLET will have lower prefetching accuracy.

RnR can achieve on an average of 97.18% of prefetching accuracy. This is because the replay iteration will have exactly the same miss sequence as the record iteration. By matching prefetching speed with demand access, most of the prefetched data will be used before it is evicted.

4) Effectiveness of replay timing control: RnR issues timely prefetches by matching the number of prefetches per window (Section V-C). As shown in the Figure 10, replay without the window-based control cannot improve performance due to the mismatch of prefetch timing with the demand access (Figure 5 (b)). After applying window control (Figure 5 (c)), RnR prefetch one window ahead without evicting prefetched

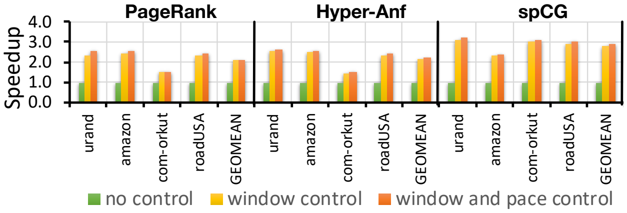


Fig. 10. Effectiveness of replay timing control.

blocks before uses. Therefore, the performance is improved significantly (2.31×). For most of the evaluated workload, window control is already good enough for controlling the prefetching speed.

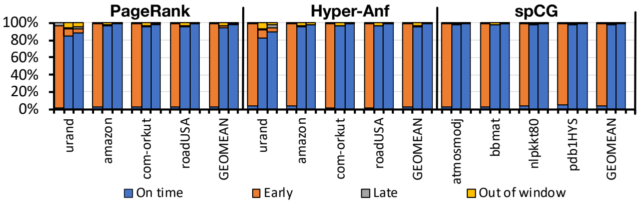


Fig. 11. Prefetch timeliness breakdown (Left bar: no control, middle bar: window control, right bar: window+pace control).

5) Timeliness: Besides coverage and accuracy, timeliness is also essential to the effectiveness of prefetching. Prefetching too early will cause prefetched data to be evicted out of the cache before it is needed. On the other hand, prefetching too late can not help to hide the miss penalty. To understand the timeliness of the proposed RnR prefetcher, we divide the total prefetched data into four categories: 1) on time prefetch, 2) early prefetch (prefetches that are demanded in the corresponding window but are evicted when the accesses arrive), 3) late prefetch (prefetches that are demanded in the corresponding window but are issued later than the accesses arrive L2), 4) out of the window (prefetches that are not demanded in the corresponding window). All of those four categories sum up to the total issued prefetches. As shown in Figure 11, most of the applications can match the prefetch speed perfectly with the demand access speed. Only two graph algorithms with urand show 7-8% of prefetches that are either early or late. This is because the off-chip demand access may overtake or delay the prefetch requests since the demand accesses are prioritized over prefetches. Applying pace control on top of window control does not significantly improve the prefetching timeliness. Only for graph algorithms with urand as input, pace control slightly reduce early prefetch by 3-4%.

6) Record Iteration Overhead: The first recorded iteration will have some performance degradation for metadata management. Given the fact that the write traffic is not on the critical path of the applications, the latency of writing metadata back to memory can be largely hidden using non-temporal (overwriting) stores. The evaluated memory system uses a write queue draining policy, which prioritizes a demand read over the write. Based on our observation, the PageRank with urand incur the largest performance degradation with only 1.75% of IPC lower than the baseline. This is because the total amount of writing traffic depends on the size of metadata. Since the synthetic graph has the highest miss rate, the writing overhead for this application is the highest. On average, the record iteration cost 1.02% of performance slow down as compared to the baseline with no prefetcher.

7) Additional Off-chip Traffic: The additional off-chip traffic is determined by $TotalPrefetch \times (1 - Accuracy) + MetadataTraffic$. On an average, bingo, SteMS, nextline, and MISB generate more than 2× of the total prefetch requests as compared to RnR. The high accuracy of RnR is the main reason for relatively low additional off-chip traffic. Metadata traffic is the main source of RnR's additional off-chip traffic. Accessing to metadata is streamed and has good spatial locality, which can efficiently utilize DRAM row buffer and memory internal parallelism. Next-line, bingo, SteMS, MISB, DROPLET, RnR, and RnR-Combined add on average of 45.2%, 67.1%, 58.4%, 19.7%, 12.2%, 12.0%, and 27.6% of additional off-chip traffic respectively. DROPLET and the RnR prefetcher add a similar amount of additional traffic. DROPLET adds additional traffic because of the inaccurate prefetching; whereas the RnR prefetcher adds extra traffic due to metadata prefetching. Inaccurate prefetching may offset the performance gains by polluting the cache space. However, the metadata are not stored in cache. MISB also stores metadata off-chip, and uses on-chip cache storage to reduce metadata traffics. The reason why MISB has a little higher off-chip traffics is also because of lower prefetching accuracy. For graph algorithms with urand as input, the metadata overhead is higher as compared to other inputs, hence incurs higher additional traffic using RnR as compared to MISB and DROPLET.

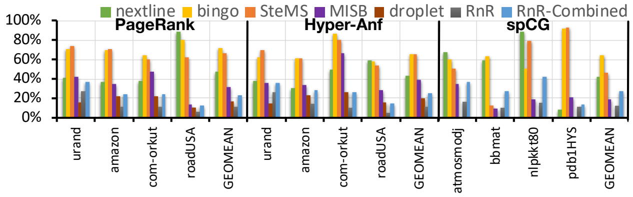


Fig. 12. Additional off-chip traffic.

B. Hardware Overhead

RnR prefetcher needs architectural state registers (Section IV), internal state registers (Section V), and a modest amount of control logic per core. The total hardware overhead for RnR prefetcher is less than 1KB for each core ($2.7E^{-3}mm^2$), which consumes less than 0.01% of the total on-chip area (46.19 mm^2).

C. Storage Overhead

RnR prefetcher requires a sequence table for the recorded miss sequence and a window division table for replay timing control. As compared to the sequence table, the window division table is much smaller as it only needs to store one word per window (thousands of addresses per window in sequence table) Therefore, most of the storage overhead comes from the miss sequence table.

The total storage overhead for each of the application and inputs depend on how large the inputs are and the data locality. For example, PageRank with the roadUSA input has good spatial/temporal locality and would require 7.64% of storage overhead of the original input size; whereas PageRank with the urand input needs 22.43% of the storage overhead due to its poor locality. Hyper-Anf with the amazon input has a 4% more storage overhead as compared to PageRank with the same input. This is because Hyper-Anf has a higher miss ratio. On average, RnR requires 12.1%, 13.0%, and 11.58% of the storage overhead for PageRank, spCG, and Hyper-Anf respectively (Figure 13).

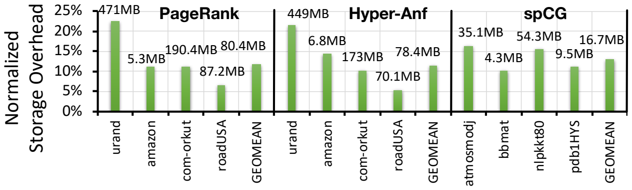


Fig. 13. Metadata storage overhead of sequence table and window division table (normalized to input dataset sizes).

D. Window Size

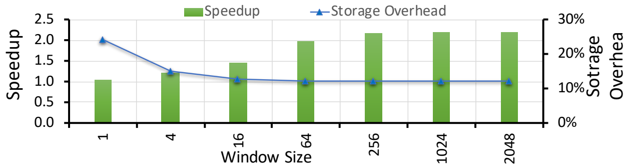


Fig. 14. Average speedup and storage for different window sizes.

The window size defines the granularity for RnR prefetcher to match the prefetch aggressiveness with the program execution (Section V-C). The smaller the window is, the more frequent the RnR prefetcher can adjust prefetching frequency. However, the smaller the window, the larger the *Window Division Table*. Because the RnR prefetches into the private L2, the window size should not exceed the half of L2 cache size. We observed that window size between 64 to 2048 cache lines achieves similar speedup as well as storage overhead (Figure 14). Window size below 64 will significantly reduce the effectiveness of RnR prefetching and increase storage overhead.

VIII. Related Work

To understand the differences of the proposed prefetcher as compared to prior work, Table IV summarizes the key design decisions made by four most related designs.

Temporal prefetchers [8], [25], [38], [58], [59] capture the irregular access pattern by memorizing the temporal address correlations. It is effective for pointer chasing based applications. Because the correlation information grows in proportion to the application's memory footprint, temporal prefetchers normally requires a large amount of metadata to achieve good performance. MISB [59] stores the metadata off-chip and prefetches them to a near core location before it is needed. MISB needs to lookup the metadata to find the corresponding address to prefetch, which requires 49KB of on-chip storage

TABLE IV
COMPARISON TO OTHER IRREGULAR HARDWARE PREFETCHERS.

Prefetcher Design	Prefetching Target Properties and Applications	What Structure to Prefetch	How to Generate Prefetch Addresses	When to Prefetch	System Overhead
RnR (Proposed)	Repeating patterns e.g. iterative solver, iterative graph kernel	Defined by software	Record and replay	Software assist, replay timing control mechanism	Hardware:less than 1KB Software: light interface Storage: 12% input
DROPLET [10]	Indirect memory access e.g. graph algorithms	Edge and vertex data	Compute based on dependency	Edge access or refill	Hardware: 12KB Software: light interface
IMP [60]	Indirect memory access, e.g. applications with SpMV kernel, graph algorithms	Index and indirect data	Compute based on dependency	Index data access	Hardware: 0.7KB
Bingo [9]	Repeating spatial patterns, e.g. OLTP, DSS	No constraint	Predict based on history	Cache access	Hardware: 119KB
MISB [59]	Repeating temporal patterns e.g. database, caching	No constraint	Predict based on history	Cache miss	Hardware: 49KB Storage: 8% input

for metadata caching. RnR does not requires metadata lookup and only neeeds 1KB per core. In terms of prefetching effectiveness, there are three reasons why RnR outperforms MISB: 1) MISB is designed for applications with common temporal patterns. As discussed in Section II , using MISB with PC localization is hard to differentiate similar temporal sequences. However, similar temporal sequences is commonly exists in graph algorithms (*e.g.*, traversing nodes within a cluster). 2) MISB can only prefetch the next few structural addresses (MISB uses a maximum prefetch degree of eight); whereas RnR can start prefetch one window ahead (up to 2048 cachelines). 3) MISB is trained at arbitrary application phase and data, which is difficult to filter out irrelevant accesses.

Spatial prefetchers [9], [47], [53] observe patterns within a spatial region, which are suitable for high-end server applications such as online transaction processing (OLTP) and decision support system (DSS) [42]. These applications normally utilize data structures that have repetitive layouts, thus recurring patterns emerge in the relative offsets of accessed data. Because the patterns are restricted within a memory region and no temporal order information is recorded, a state-of-the-art spatial prefetcher normally requires a minimal amount of metadata, which could be stored in an on-chip hardware table.

IMP [60] is designed for indirect memory accesses to follow the A[B[i]] indirect accesses, which shares some common targets with RnR. However, IMP is a purely hardware design, which generates indirect addresses by predicting the correlated index stream. This approach suffers from low prefetching accuracy and ill-timed prefetches, which leads to low miss coverage. Programmable Prefetcher [6] focused on indirect memory access (e.g, C[B[A[x]]]), where accessing array A has memory-level parallelism and prefetching array B, C does not need to wait. This technique needs compiler assistance, which offload software prefetches to programmable hardware.

DROPLET [10] is a more recent work that targets on indirect memory accesses in graph algorithms. It also has a lightweight software interface to define the targeted data structure (spatial region). DROPLET generates the addresses for an indirectly accessed vertex value by prefetching the edge array. Since DROPLET is equipped with software knowledge, it's accuracy and coverage is higher than IMP. However, the vertex data prefetching is triggered when edge data refills the DRAM read queue, which is often too late. As compared to DROPLET, RnR records the miss address patterns as well as the timing information. Although it requires additional storage space, the RnR improves both the prefetch accuracy and timeliness.

Software prefetchers [37] often achieve higher accuracy and coverage as compared to hardware prefetchers. Several related works propose inserting prefetching instructions at an appropriate place through programmer and compiler efforts. However, software prefetching schemes are difficult to migrate to a different microarchitectures because it lacks of the hardware knowledge. Furthermore, it could incur large instruction overhead, which may offset the prefetching benefits.

Compared to prior work, the **novelties and advantages** of the proposed RnR prefetcher include: 1) It provides a lightweight software interface to communicate the spatial and temporal information of the prefetch region of interest to the hardware prefetcher, which improves prefetching accuracy and miss coverage. 2) It records the private cache miss addresses and miss ratio in each prefetch window for the targeted data structure, and uses a replay timing control mechanism to achieve timely prefetches. and 3) RnR is not bound to a particular microarchitecture, it is scalable and can co-exist with other prefetchers.

IX. CONCLUSION

In this work, a novel software-assisted RnR hardware prefethcer is proposed to improve prefethching accuracy and miss coverage for applications that have long repeating irregular memory access patterns. By allowing programmer to decide when and what to record and replay, the proposed RnR prefetcher can efficiently record miss sequence of the target data structure and prefetch through replay in a timely manner.

ACKNOWLEDGMENT

This material is based upon work supported by the National Science Foundation at Lehigh University under Grant CCF-1750826, CCF-1723624, and by the Director, Office of Science, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation nor the U.S. Department of Energy.

REFERENCES

[1] "The 3rd data prefetching championship," https://dpc3.compas.cs.stonybrook.edu/.

[2] "8gb: x4, x8, x16 ddr4 sdram features," micron Technology, Inc. https://www.micron.com/-/media/client/global/documents/products/data-sheet/dram/ddr4/8gb_ddr4_sdram.pdf.

[3] "Champsim," https://github.com/ChampSim/ChampSim.

[4] "Intel i7-6700 (skylake), 4.0 ghz (turbo boost), 14 nm." https://www.intel.com/content/www/us/en/processors/core/desktop-6th-gen-core-family-datasheet-vol-1.html.

[5] 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.

[6] S. Ainsworth and T. M. Jones, "An event-triggered programmable prefetcher for irregular workloads," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 578–592. [Online]. Available: https://doi.org/10.1145/3173162.3173189

[7] 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.

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

[9] 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)*. IEEE, 2019, pp. 399–411.

[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] M. Besta, M. Podstawski, L. Groner, E. Solomonik, and T. Hoefler, "To push or to pull: On reducing communication and synchronization in graph computations," in *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, 2017, pp. 93–104.

[13] P. Boldi, M. Rosa, and S. Vigna, "Hyperanf: Approximating the neighbourhood function of very large graphs on a budget," in *Proceedings of the 20th International Conference on World Wide Web*, ser. WWW '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 625–634. [Online]. Available: https://doi.org/10.1145/1963405.1963493

[14] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," 1998.

[15] "Cadence genus synthesis solution," Cadense, https://www.cadence.com/content/dam/cadence-www/global/en_US/documents/tools/digital-design-signoff/genus-synthesis-solution-ds.pdf.

[16] R. Chandra, L. Dagum, D. Kohr, R. Menon, D. Maydan, and J. McDonald, *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[17] F. Darema, D. A. George, V. A. Norton, and G. F. Pfister, "A single-program-multiple-data computational model for epex/fortran," *Parallel Computing*, vol. 7, no. 1, pp. 11–24, 1988.

[18] J. Davidson, B. Liebald, J. Liu, P. Nandy, T. Van Vleet, U. Gargi, S. Gupta, Y. He, M. Lambert, B. Livingston *et al.*, "The youtube video recommendation system," in *Proceedings of the fourth ACM conference on Recommender systems*, 2010, pp. 293–296.

[19] 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.

[20] J. Dongarra, M. A. Heroux, and P. Luszczek, "High-performance conjugate-gradient benchmark: A new metric for ranking high-performance computing systems," *The International Journal of High Performance Computing Applications*, vol. 30, no. 1, pp. 3–10, 2016.

[21] J. Doweck, "White paper inside intel® core™ microarchitecture and smart memory access," *Intel Corporation*, vol. 52, pp. 72–87, 2006.

[22] C. Du and X.-H. S. Sun, "Mpi-mitten: Enabling migration technology in mpi," in *Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06)*, vol. 1. IEEE, 2006, pp. 11–18.

[23] N. J. et al., "Adept deliverable d2.3 - updated report on adept benchmarks," 2015.

[24] W. Gropp, W. D. Gropp, E. Lusk, A. D. F. E. E. Lusk, and A. Skjellum, *Using MPI: portable parallel programming with the message-passing interface*. MIT press, 1999, vol. 1.

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

[26] N. Jammula, M. Qureshi, A. Gavrilovska, and J. Kim, "Balancing context switch penalty and response time with elastic time slicing," in *2014 21st International Conference on High Performance Computing (HiPC)*. IEEE, 2014, pp. 1–10.

[27] D. A. Jiménez and C. Lin, "Dynamic branch prediction with perceptrons," in *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*. IEEE, 2001, pp. 197–206.

[28] U. Kang, D. H. Chau, and C. Faloutsos, "Mining large graphs: Algorithms, inference, and discoveries," in *2011 IEEE 27th International Conference on Data Engineering*, 2011, pp. 243–254.

[29] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998.

[30] C. N. Keltcher, K. J. McGrath, A. Ahmed, and P. Conway, "The amd opteron processor for multiprocessor servers," *IEEE Micro*, vol. 23, no. 2, pp. 66–76, 2003.

[31] Y. Kozawa, T. Amagasa, and H. Kitagawa, "Gpu-accelerated graph clustering via parallel label propagation," in *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, ser. CIKM '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 567–576. [Online]. Available: https://doi.org/10.1145/3132847.3132960

[32] J. Leskovec and R. Sosič, "Snap: A general-purpose network analysis and graph-mining library," *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 8, no. 1, p. 1, 2016.

[33] C. Li, C. Ding, and K. Shen, "Quantifying the cost of context switch," in *Proceedings of the 2007 workshop on Experimental computer science*, 2007, pp. 2–es.

[34] C. C. Liu, I. Ganusov, M. Burtscher, and S. Tiwari, "Bridging the processor-memory performance gap with 3d ic technology," *IEEE Design & Test of Computers*, vol. 22, no. 6, pp. 556–564, 2005.

[35] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," *Acm sigplan notices*, vol. 40, no. 6, pp. 190–200, 2005.

[36] P. Michaud, "Best-offset hardware prefetching," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016, pp. 469–480.

[37] 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.

[38] K. J. Nesbit and J. E. Smith, "Data cache prefetching using a global history buffer," in *10th International Symposium on High Performance Computer Architecture (HPCA'04)*. IEEE, 2004, pp. 96–96.

[39] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web." Stanford InfoLab, Tech. Rep., 1999.

[40] J. Park, M. Smelyanskiy, K. Vaidyanathan, A. Heinecke, D. D. Kalamkar, X. Liu, M. M. A. Patwary, Y. Lu, and P. Dubey, "Efficient shared-memory implementation of high-performance conjugate gradient benchmark and its application to unstructured matrices," in *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2014, pp. 945–955.

[41] S. H. Pugsley, Z. Chishti, C. Wilkerson, P. Chuang, R. L. Scott, A. Jaleel, S. Lu, K. Chow, and R. Balasubramonian, "Sandbox prefetching: Safe run-time evaluation of aggressive prefetchers," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, 2014, pp. 626–637.

[42] P. Ranganathan, K. Gharachorloo, S. V. Adve, and L. A. Barroso, "Performance of database workloads on shared-memory systems with out-of-order processors," in *Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, 1998, pp. 307–318.

[43] A. D. Robison, "Cilk plus: Language support for thread and vector parallelism," *Talk at HP-CAST*, vol. 18, p. 25, 2012.

[44] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-stream: Edge-centric graph processing using streaming partitions," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013, pp. 472–488.

[45] B. T. Sander, W. A. Hughes, S. P. Subramanian, and T.-C. Tan, “Stride based prefetcher with confidence counter and dynamic prefetch-ahead mechanism,” May 27 2003, uS Patent 6,571,318.

[46] T. Sherwood, S. Sair, and B. Calder, “Predictor-directed stream buffers,” in *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, 2000, pp. 42–53.

[47] M. Shevgoor, S. Koladiya, R. Balasubramonian, C. Wilkerson, S. H. Pugsley, and Z. Chishti, “Efficiently prefetching complex address patterns,” in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2015, pp. 141–152.

[48] J. Shun and G. E. Blelloch, “Ligra: a lightweight graph processing framework for shared memory,” in *ACM Sigplan Notices*, vol. 48. ACM, 2013, pp. 135–146.

[49] J. Shun, F. Roosta-Khorasani, K. Fountoulakis, and M. W. Mahoney, “Parallel local graph clustering,” *Proceedings of the VLDB Endowment*, vol. 9, no. 12, pp. 1041–1052, 2016.

[50] J. E. Smith and W.-C. Hsu, “Prefetching in supercomputer instruction caches,” in *Proceedings of the 1992 ACM/IEEE conference on Supercomputing*. IEEE Computer Society Press, 1992, pp. 588–597.

[51] A. Sodani, R. Gramunt, J. Corbal, H.-S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y.-C. Liu, “Knights landing: Second-generation intel xeon phi product,” *Ieee micro*, vol. 36, no. 2, pp. 34–46, 2016.

[52] S. Somogyi, T. F. Wenisch, A. Ailamaki, and B. Falsafi, “Spatio-temporal memory streaming,” in *ACM SIGARCH Computer Architecture News*, vol. 37. ACM, 2009, pp. 69–80.

[53] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, “Spatial memory streaming,” *ACM SIGARCH Computer Architecture News*, vol. 34, no. 2, pp. 252–263, 2006.

[54] J. E. Stine, I. Castellanos, M. Wood, J. Henson, F. Love, W. R. Davis, P. D. Franzon, M. Bucher, S. Basavarajaiah, J. Oh *et al.*, “Freepdk: An open-source variation-aware design kit,” in *2007 IEEE international conference on Microelectronic Systems Education (MSE’07)*. IEEE, 2007, pp. 173–174.

[55] Tien-Fu Chen and Jean-Loup Baer, “Effective hardware-based data prefetching for high-performance processors,” *IEEE Transactions on Computers*, vol. 44, no. 5, pp. 609–623, 1995.

[56] S. Venkataraman, E. Bodzsar, I. Roy, A. AuYoung, and R. S. Schreiber, “Presto: distributed machine learning and graph processing with sparse matrices,” in *Proceedings of the 8th ACM European Conference on Computer Systems*, 2013, pp. 197–210.

[57] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, “Optimization of sparse matrix-vector multiplication on emerging multicore platforms,” in *SC’07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*. IEEE, 2007, pp. 1–12.

[58] H. Wu, K. Nathella, J. Pusdesris, D. Sunwoo, A. Jain, and C. Lin, “Temporal prefetching without the off-chip metadata,” in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 996–1008.

[59] H. Wu, K. Nathella, D. Sunwoo, A. Jain, and C. Lin, “Efficient metadata management for irregular data prefetching,” in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 449–461.

[60] X. Yu, C. J. Hughes, N. Satish, and S. Devadas, “Imp: Indirect memory prefetcher,” in *Proceedings of the 48th International Symposium on Microarchitecture*. ACM, 2015, pp. 178–190.