Software Prefetching for Indirect Memory Accesses

Aditya Kumar Bej
Department of Computer Science
University of California, Davis
Davis, California
akbej@ucdavis.edu

Prem Gorde
Department of Computer Science
University of California, Davis
Davis, California
pkgorde@ucdavis.edu

***Abstract*—In the field of computing, complex software algorithms play a crucial role in powering the backend systems of various applications. These applications include graph processing for social networks like Facebook's TAO system, scientific simulations with GROMACS, advanced gaming engines from Unreal and Unity, and database management systems such as MongoDB and Apache Cassandra. These accesses, involving navigation through data structures like trees and graphs, can cause significant delays due to the unpredictable nature of memory access patterns, impacting system performance.**

Conventional prefetching strategies, predominantly hardware-oriented, often struggle with such irregular access patterns, thereby compromising their ability to efficiently prefetch required data. Although software prefetching emerges as a viable alternative, it typically requires programmers to manually insert special instructions into the code, a process that is both time-consuming and prone to errors.

This paper explores an automated approach to software prefetching, specifically targeting indirect memory accesses in complex and data-heavy applications. By automating the process, we aim to save time and reduce errors, leading to improvements in overall system performance. We present an analysis of the current state of automated software prefetching, propose improvements over existing solutions, and benchmark these against the latest versions of popular sparse workloads. This advancement could mark a significant step forward in high-performance computing.

***Index Terms*—Prefetching, Cache, Compiler Analysis, Automated Software Prefetching**

I. Introduction

In the evolving landscape of computing, especially in large-scale computing, the efficiency of memory access patterns plays a pivotal role in determining the overall performance of software applications. As modern computational workloads become increasingly complex [6] [13], particularly with the growing prevalence of data-intensive tasks, the challenge of efficiently managing memory accesses has come to the forefront. One major challenge to this aspect is handling of indirect memory accesses (IMAs) which is a common scenario in various high-level computations, ranging from sophisticated 3D rendering used in gaming and simulations. IMAs occur when the memory address to be accessed is not explicitly known at compile-time but instead computed dynamically during program execution. This makes it difficult for hardware prefetchers to accurately predict future memory accesses.

Software prefetching [5] for IMAs attempts to address this challenge by leveraging compiler analysis [8] and optimization techniques to anticipate future memory accesses based on a program context and access patterns. This proactive approach can significantly improve memory performance by preloading data into the processor cache before it is actually needed. Existing approaches for stride and linked-list access patterns [14] have limited effectiveness due to factors such as hardware efficiency and inherent lack of parallelism. However indirect memory accesses display abundant parallelism, making them ideal candidates for automated prefetching. The widespread adoption of software prefetching faces a few major hurdles. While manual prefetching offers finer-grained control, it is a time-consuming and error-prone process [9]. Developers need to manually identify and insert prefetch instructions, requiring in-depth understanding of the program's memory access patterns and the target architecture. And apart from manual prefetching, there are limited automatic techniques where existing compiler-based techniques [4] for automatically identifying and prefetching IMAs are limited in their scope and effectiveness due to factors such as not being tested on latest sparse workloads. They often require specific access patterns and may not be able to handle complex program behavior.

In this paper, we conduct a comprehensive study of existing prefetching techniques that employ both software and hardware strategies, particularly focusing on sparse data structures. We evaluate several promising ideas from these studies to enhance our compiler-based prefetching algorithm. Furthermore, we analyze the limitations of existing solutions and explore intuitive approaches to address these challenges. We also place special emphasis on testing our automated software prefetching algorithm with the latest versions of sparse workloads, which are common in numerous applications. Our testing environment utilizes the latest software versions, including Ubuntu 22.04 and LLVM with Clang 14.0. This modern setup ensures that our testing accurately reflects the latest computing environments, providing a relevant and updated comparison with earlier research.

II. Problem Definition

The primary challenge addressed in this study is the optimization of memory access in software applications that extensively utilize indirect memory accesses (IMAs). These IMAs are a common feature in a range of high-performance computing tasks, yet their dynamic nature makes prefetching data into the processor cache a complex endeavor. Traditional

hardware prefetching methods are often ineffective due to their inability to anticipate the dynamically computed memory addresses that characterize IMAs. This leads to a significant increase in memory access time, thereby impeding overall application performance.

Current software-based prefetching techniques, while offering a potential solution, face limitations in terms of their scope and efficiency. The manual insertion of prefetch instructions, although precise, is a laborious and error-prone process, requiring extensive knowledge of both the application's memory usage and the underlying hardware architecture. Automated methods, on the other hand, are still in their nascent stages, with limited effectiveness in dealing with the complexities of modern software environments, particularly in the context of sparse data structures. Related work on automated prefetching techniques [1] [4] utilize older versions of sparse workloads and outdated testing environments.

III. Related work and its limitations

Software prefetching has garnered considerable attention in prior research, and our work offers a comprehensive overview of various techniques. This includes an analysis of their performance impact, methods for automating prefetch instruction insertion, a combination of some efficient hardware and software based prefetching techniques. We also note down the limitations and bottleneck for these studies which helps in evaluating gaps in current research.

A. IMP: indirect memory prefetcher [2]

IMP, implemented by Xu. et al, very closely tackles the problem we are approaching in our study, but propose an alternative solution track which has also given good results. IMP is a hardware implementation which deals specifically with irregular memory accesses in sparse data structures with an efficient hardware indirect memory prefetcher to capture access patterns and hide latency. Specifically, this method is able to target irregular memory accesses of non-zero elements in a sparse matrix, which of the nature A[B[i]]. Along with this, they have also kept the problem with cachelines and sparse data structures in mind, and proposed a partial cacheline accessing mechanism, since not all the data in a given cacheline is needed at the same time. Their partial cacheline accessing mechanism is able to fetch only the important parts of the block required. According to their study, they have been able to achieve up to 2.3x speedup, and another 9.4% with the partial cacheline accessing.

The main drawback of this approach stems from its hardware-based nature. Since it is a hardware implementation, it is very difficult to customise the data access patterns, which are very easily modifiable in software. Similarly, integrating IMP with different micro-architectures and system configurations is not feasible. It's specific need of hardware architectures reduces its portability. Also, its own complexity in implementation makes it difficult to implement and re-use and/or modify.

B. Array Tracking Prefetcher: Informed Prefetching for Indirect Memory Accesses [3]

Cavus et al. propose ATP, the Array Tracking Prefetcher, which tracks array-based indirect memory accesses using a novel combination of software and hardware. It uses pre-inserted configurations from the programmer and the compiler to pass data structure traversal knowledge. The big advantage of this implementation is that it achieves a speedup of 2.17x compared to a single-core system without prefetching, and a speedup of 1.84x over conventional software and over 1.32x over only hardware prefetching [3].

The downside of this implementation is that the special metadata being inserted in the beginning must be done by the programmer. From there, there is a large amount of complexity to be overcome in order to finish the implementation due to it being both hardware and software combined. This combined implementation also makes it hard for the mechanism to adapt to runtime changes in the memory access patterns, especially in highly variable or unpredictable workloads. And since we want to focus on such workloads, this implementation does not provide as much merit.

C. APT-GET: profile-guided timely software prefetching [4]

One of the most recent developments in the software prefetching realm, APT-GET proposes a solution beyond static compiler-based prefetching mechanism. APT-GET is designed to identify and pre-load future memory accesses for irregular access patterns. It does so by calculating optimal prefetching distance and the best site for optimal prefetching injection. The most novel part of this study is that this prefetching injection is automated, and also provides variable prefetch-distance and injection sites. This facilitates easier integration into development workloads.

One disadvantage of the paper is that it is only tested on older sparse workloads, which do not provide a good representation of modern workloads, where such techniques are needed. Also, since this implementation of APT-GET specifically relies on Intel's Last Branch Record (LBR) hardware support system, it is hard to speculate the performance of this software without it, or with some other implementation. Testing of this implementation on different systems and different, more complex workloads, or even real-time applications needs to be studied to evaluate its performance metrics more accurately.

IV. Plan for Improvement

Our work for this paper is the following

- To focus exclusively on sparse workloads only with software prefetching.
- Test on datasets relating to sparse workloads used in modern real world applications such as in modern graph workloads and various implementations of generating the graph structure, Integer and Conjugate Gradient functions, and random accesses in High Performance Computing systems.

- Make efforts on good manual prefetching for a comparative analysis with automated prefetching. This involves studying the source code of the workloads in detail and identifying the most appropriate injection sites.
- To focus on improving and extending the functionalities of existing automated software prefetch generation algorithms [1, 4] in the most recent versions of the technology stack such as LLVM, Clang, Ubuntu in order to drastically reduce the inconvenience and time to execute and test even more sparser workloads when compared to manual prefetching.
- Our workloads are more sparser compared to previous related works [1, 4] since we use modified parameters which affect the sparsity of the workloads. The nuances behind these benchmark parameters are explained in Subsection IV-A.

A. Choosing the right workload

1) **Graph 500**

The Graph500 workload was specifically designed to measure the performance of systems data-intensive graph analytics workloads. The workload performs a breadth-first search on a generated Kronecker graph [10]. A Kronecker graph is a type of graph generated by the Kronecker product, which is used primarily in network analysis, particularly for modeling and simulating large-scale networks like social networks, web graphs, and biological networks.

This workload has been a popular choice used by other software prefetching papers [1,4]. The before-mentioned papers use only the Sequential compressed-Sparse-Row implementation as the workload for testing. But we have chosen to test on 3 different implementations of the graph500 workload as we can get broader performance insights due to the various data structures used in the implementation. This overall allows us to comprehensively benchmark the effectiveness of the software prefetching algorithm against different approaches of the graph implementation. From our research, we have not seen any other paper who have tested against multiple implementations of graph500 in the context of automated software prefetching. The benchmark parameters also differ compared to the previous papers as we are using a higher scale and edge factor with modified initiator parameters (A, B, C, D) for generating our graph for each implementation.

a) Sequential List-based Implementation
b) Sequential Compressed-Sparse-Row Implementation
c) OpenMP Compressed-Sparse-Row Implementation

1.A) Graph 500 Sequential List-based Implementation

- **Benchmark** - Graph500 [7]
- **Overview** - The seq-list workload [7] involves a sequential list-based implementation for processing large-scale graph data. This method represents the graph as a list of edges or vertices, which the algorithm sequentially traverses. The focus is on operations like graph traversal, search, and pathfinding, common in many graph analytics tasks.
- **Use case** - This implementation is particularly useful in environments where parallel computing resources are limited or unavailable. It serves as a baseline for evaluating more complex, parallel implementations of graph algorithms. It's representative of scenarios in traditional single-threaded applications dealing with graph data, such as certain types of database queries, network analysis, and basic graph computations.
- **Analysis overview** - The performance analysis of this workload would involve measuring the efficiency of list-based graph traversal, particularly in terms of time complexity and memory access patterns. The sequential nature of the algorithm offers insights into the baseline performance of graph processing tasks without parallel optimizations.
- **Benchmark parameters** - Scale = **20** (2^{20} or approximately 1,048,576 vertices) ; Edge factor = **16** ; Initiator Parameters **A=0.25; B=0.30; C=0.30; D=0.05**
- **Parameter information**:
  - **Scale** - SCALE is a parameter in the Graph500 benchmark that determines the size of the graph by specifying the logarithm base two of the number of vertices. An increase in SCALE leads to an exponential growth in the number of vertices, resulting in a larger graph with a more significant memory footprint. This could cause more cache misses, especially if prefetching is not optimized for larger data sets. Conversely, decreasing SCALE results in a smaller graph with fewer vertices, potentially enhancing cache efficiency and prefetching effectiveness due to the reduced data size.
  - **Edge Factor** - Edge factor is the ratio of the graph's edge count to its vertex count, essentially representing half the average degree of a vertex in the graph. Increasing the edge factor leads to a denser graph with more edges per vertex, complicating graph traversal algorithms and potentially leading to irregular memory access patterns, challenging the effectiveness of prefetching. Decreasing the edge factor results in a sparser graph, simplifying memory access patterns and possibly making prefetching more efficient.
  - **Initiator Parameters (A,B,C,D)** - The initiator parameters (A, B, C, D) in the Graph500 benchmark define the probabilities of placing an edge in one of four partitions of the adjacency

matrix during graph generation. Altering these parameters changes the graph's structure, impacting memory access patterns during processing. For example, a higher value of A increases the likelihood of edges within the same partition, leading to more localized memory accesses that might be better suited for prefetching. In contrast, a more uniform distribution among these parameters might result in random access patterns, posing a greater challenge for prefetching strategies.

1.B) Graph 500 Sequential Compressed-Sparse-Row Implementation

- **Benchmark** - Graph500 [7]
- **Overview** - The seq-csr workload uses a sequential approach with the compressed-sparse-row (CSR) format for graph representation. CSR is efficient for storing and accessing sparse matrices, which is a common representation for graphs, especially those with large numbers of vertices and edges.
- **Use case** - This implementation is ideal for analyzing the performance of graph algorithms on sparse graphs, particularly where memory efficiency is crucial. It's representative of applications like social network analysis, web graph processing, and sparse matrix computations in scientific computing.
- **Analysis overview** - Analyzing this workload focuses on the efficiency of the CSR format in terms of memory usage and access, as well as the computational complexity of graph algorithms in a sequential computing environment. It provides a benchmark for the performance of sparse graph processing without parallelization.
- **Benchmark parameters** - Scale = **20** (2^{20} or approximately 1,048,576 vertices) ; Edge factor = **16** ; Initiator Parameters **A=0.25; B=0.30; C=0.30; D=0.05**

1.C) Graph 500 OpenMP Compressed-Sparse-Row Implementation

- **Benchmark** - Graph500 [7]
- **Overview** - The omp-csr workload extends the CSR implementation with parallel processing using OpenMP [11], a widely-used parallel programming model. This approach leverages multi-core processors to handle graph processing tasks more efficiently.
- **Use case** - This implementation is suited for environments where parallel processing resources are available. It's particularly relevant for multi-threaded applications in high-performance computing, where efficient parallel processing of large-scale, sparse graphs is required.
- **Analysis overview** - Performance analysis here involves evaluating the effectiveness of parallel optimizations in the CSR format for graph processing. The focus is on measuring improvements in computational speed, efficiency of memory access in a multi-threaded context, and scalability of the algorithm across multiple processor cores.
- **Benchmark parameters** - Scale = **20** (2^{20} or approximately 1,048,576 vertices) ; Edge factor = **16** ; Initiator Parameters **A=0.25; B=0.30; C=0.30; D=0.05**

We have also decided to test a few other following workloads. The intuition for choosing each workload is mentioned alongside.

2) **NAS-Integer Sort**
   - **Benchmark** - NAS Parallel Benchmarks [6]
   - **Overview** - Integers are sorted using a bucket sort, walking an array of integers and resulting in array-indirect accesses to increment the bucket of each observed value.
   - **Use case** - Representative of computational fluid dynamics workloads.
   - **Analysis overview** - Tested with the software prefetching algorithm by inserting software prefetches in the loop that updates the bucket count,This is done by looking ahead in the main array and issuing prefetch commands based on the indices derived from these upcoming values.
   - **Benchmark parameters** - Class B ; Size = **33554432** ; Number of iterations - **10**
   - **Parameter information**:
     - **Size** - Increasing the Size parameter results in a larger array of integers to sort, which increases the computational complexity and memory usage. It challenges the efficiency of the sorting algorithm, especially in terms of memory access patterns and cache utilization
     - **Number of Iterations** - Increasing the number of iterations will repeat the sorting task more times, amplifying any performance characteristics or bottlenecks of the algorithm.
   - **Benchmark Version** - We are using the latest version of NAS-IS (3.3-Open MP) from the NAS Parallel Benchmarks. The other papers [1, 4] dealing with this workload used older versions (2.3)
3) **NAS-Conjugate Gradient**
   - **Benchmark** - NAS Parallel Benchmarks [6]
   - **Overview** - Performs eigenvalue estimation on sparse matrices. The sparse matrix multiplication exhibits an array-indirect pattern.
   - **Use case** - Design is typical of unstructured grid computations
   - **Analysis overview** - The array-indirect pattern allows the software prefetches to be inserted based on the benchmark's NZ matrix which stores non-zeros,

using the stored indices of the dense vector it points to. The irregular access is on a smaller dataset than Integer Sort, meaning it is more likely to fit in the L2 cache and is less challenging for a TLB system.

- **Benchmark parameters** - Class C ; Size = **150000** ; Non-zero elements = **15** ; Number of iterations = **75** ; Eigenvalue shift - **110.0**
- **Parameter information**:
  - **Size** - Increasing the size makes the problem larger and more computationally intensive, resulting in more memory usage and possibly necessitating more efficient memory access strategies
  - **Non-zero elements** - Specifies the average number of non-zero elements per row in the matrix. Increasing this makes the matrix denser, which could lead to more regular memory access patterns, and decreasing makes the matrix sparser, potentially leading to more irregular memory access patterns.
  - **Number of iterations** - Sets the number of iterations in the conjugate gradient algorithm. Increasing it extends the computation, enhancing the impact of memory access patterns on overall performance and decreasing shortens the computation, which might reduce the effectiveness of prefetching.
  - **Eigenvalue shift** - This is used to modify the matrix eigenvalues, which can affect the condition number and convergence properties of the algorithm. Adjusting this can impact the numerical behavior of the algorithm but does not directly influence the computational workload or memory access patterns.
- **Benchmark Version** - We are using the latest version of NAS-CG (3.4-Open MP) from the NAS Parallel Benchmarks. The other papers [1, 4] dealing with this workload used older versions (2.3)

4) **Camel - Multi Hashing**

- **Benchmark** - Custom Benchmark developed from NAS-IS [1]
- **Overview** - This benchmark measures the performance of a program that traverses a hash chain with varying depths determined by NUMHASH. The program calculates the sum of values through the chain, where each value is accessed indirectly through a hash function.
- **Use case** - This benchmark can be used to evaluate the effectiveness of prefetching techniques for indirect memory accesses, particularly in scenarios involving nested hashing and deep memory dependency chains.
- **Analysis overview** - The benchmark analyzes the time taken to traverse the hash chain and calculate the sum. This time is then used to evaluate the impact of prefetching on memory access performance and overall program execution speed.
- **Benchmark parameters** - MAX KEY = **33554432** ; NUMHASH = **20** ; PREFETCH = **0/1 (Boolean)**
- **Parameter information**:
  - **MAX KEY** - Increasing the size makes the problem larger and more computationally intensive, reulting in more memory usage and possibly necessitating more efficient memory access strategies
  - **NUMHASH** - Specifies the average number of non-zero elements per row in the matrix. Increasing this makes the matrix denser, which could lead to more regular memory access patterns, and decreasing makes the matrix sparser, potentially leading to more irregular memory access patterns.
  - **PREFETCH** - Sets the number of iterations in the conjugate gradient algorithm. Increasing it extends the computation, enhancing the impact of memory access patterns on overall performance and decreasing shortens the computation, which might reduce the effectiveness of prefetching.

B. Manual Prefetching Analysis

This analysis focuses on the manual injection of prefetch instructions as a method of optimization and compares it with automated prefetching techniques. For each of the benchmarks, we analyzed the source code to understand the algorithm and identify injection sites for the prefetch instructions. We discuss the high level overview of these sites for each of the benchmarks

Graph 500

For every graph 500 based implementation, we did an manual in-order prefetching and out-of-order prefetching. In-order prefetching involves prefetching data that is expected to be accessed in the near future, based on the current memory access pattern. The idea is to reduce cache misses by having data ready in the cache before it is actually needed. Out-of-order prefetching is a more complex strategy where data is prefetched not strictly based on immediate upcoming accesses, but potentially based on expected accesses that are not immediately next in the sequence.

1) **Graph 500 Sequential List-based Implementation** Prefetch instructions were inserted in two functions named **create-graph-from-edgelist** and **make-bfs-tree**. **In-order prefetching** - As described in Algorithm 1, the focus is on the part of the function where in-order prefetching is used. The perform in-order instruction is conditionally executed to ensure it remains within the bounds of the IJ-in array. This prefetching is strategically placed within the main loop, which processes each edge in the edge list. It aims to reduce cache misses by loading the data of future elements into the cache ahead of their actual use, improving the efficiency of the graph construction. For `make_bfs_tree`, As described in Algorithm 2, the prefetch instruction is applied to prefetch elements from

the vlist and bfs-tree arrays based on the current index and PREFETCH_DISTANCE. This prefetching strategy is designed to improve cache efficiency during the BFS tree construction process. By preloading data into the cache before it is actually used, the function can reduce cache misses, leading to potentially better performance.

Algorithm 1 Graph Construction from Edge List with In-Order Prefetching

Input: Edge list IJ_in, number of edges $nedge$
Output: Graph constructed from edge list
Procedure: CREATEGRAPHFROMEDGELIST
for $k = 0$ to $nedge - 1$ **do**
  Perform in-order prefetching for future edges
  Process edge k from IJ_in (details omitted)
end for
return err

Algorithm 2 BFS Tree Construction with In-Order Prefetching

Input: Vertex list $vlist$, source vertex $srcvtx$
Output: BFS tree bfs_tree_out, maximum vertex max_vtx_out
Procedure: MAKEBFSTREE
Initialize bfs_tree_out, max_vtx_out, allocate and initialize $vlist$
while termination condition not met **do**
  Determine range for current iteration
  for k in the determined range **do**
    Perform in-order prefetching for $vlist$ and bfs_tree
    Expand BFS tree using current vertex (details omitted)
  end for
  Update loop control variables
end while
Free allocated memory
return err

Out of order prefetching - As described in Algorithm 3 for `create_graph_from_edgelist`, the part of the function where out-of-order prefetching is utilized is shown. The builtin-prefetch instruction is applied using a `prefetch_index` that is calculated to potentially jump ahead in the edge list, going beyond a simple sequential approach. This out-of-order prefetching strategy is designed to anticipate and handle non-linear or less predictable memory access patterns that can occur during graph construction. It aims to preload relevant data into the cache before it's accessed, potentially improving performance by reducing cache misses. For `make_bfs_tree`, As described in Algorithm 4, the builtin-prefetch instruction is applied with a prefetch_index that potentially jumps ahead in the vertex list in a non-sequential manner. This prefetching strategy aims to handle irregular or non-linear access patterns during the BFS tree construction. It preloads data into the cache before it's accessed, potentially improving performance in scenarios where memory access patterns are less predictable.

Algorithm 3 Graph Construction from Edge List with Out-of-Order Prefetching

Input: Edge list IJ_in, number of edges $nedge$
Output: Graph constructed from edge list
Procedure: CREATEGRAPHFROMEDGELIST
for $k = 0$ to $nedge - 1$ **do**
  Calculate prefetch index within bounds
  Perform out-of-order prefetching for future edges
  Process edge k from IJ_in (details omitted)
end for
return err

Algorithm 4 BFS Tree Construction with Out-of-Order Prefetching

Input: Vertex list $vlist$, source vertex $srcvtx$, number of vertices n
Output: BFS tree bfs_tree_out, maximum vertex max_vtx_out
Procedure: MAKEBFSTREE
Initialize bfs_tree_out, max_vtx_out, allocate and initialize $vlist$
while termination condition not met **do**
  Determine range for current iteration
  for k in the determined range **do**
    Calculate prefetch index within bounds
    Perform out-of-order prefetching for $vlist$ and bfs_tree
    Expand BFS tree using current vertex (details omitted)
  end for
  Update control variables
end while
Free allocated memory
return err

2) **Graph 500 Sequential Compressed-Sparse-Row Implementation** - **In-order prefetching** - As described in Algorithm 5, this prefetching strategy aims to optimize memory access patterns during the BFS tree construction in a graph represented using a Compressed-Sparse-Row (CSR) format. This type prefetches data in the order it will be accessed by the loop. The focus is on prefetching future entries of the vertex list $vlist$ in the current iteration, which aligns with the in-order prefetching strategy.

Algorithm 5 Pseudocode for Graph 500 CSR In-order prefetching

Input: Vertex list $vlist$, source vertex $srcvtx$, number of vertices nv
Output: BFS tree bfs_tree_out, maximum vertex max_vtx_out
Initialize bfs_tree_out and max_vtx_out
Allocate memory for vertex list $vlist$
if memory allocation fails **then**
  return -1
end if
Initialize bfs_tree_out with $srcvtx$
Initialize control variables $k1$, $k2$
while $k1 \neq k2$ **do**
  for k from $k1$ to $oldk2 - 1$ **do**
    In-Order Prefetching: Prefetch future entries of $vlist$ for iteration k
    for each adjacent vertex of v **do**
      Update BFS tree if the vertex is not visited
    end for
  end for
  Update control variables $k1$, $k2$
end while
Free allocated memory for $vlist$
return err

Out of order prefetching - As described in Algorithm 6 Instead of prefetching a fixed number of elements ahead (in-order), this implementation uses conditional prefetching based on the value of `vlist[k + 8]`. This means data is only prefetched if the element is non-zero, reducing unnecessary overhead for empty entries. An optional prefetch for `vlist[k + 12]` is included within a `#ifdef STRIDE` block. This suggests that this prefetch is likely based on a known stride pattern in the `vlist` array, potentially improving performance for specific scenarios.

3) **Graph 500 OpenMP Compressed-Sparse-Row Implementation**

In-order prefetching - As described in Algorithm 7. In the `gather_edges` function, OpenMP is utilized (with the 'OMP("omp parallel")' directive) to parallelize the processing of graph edges. This parallelization allows for simultaneous processing of multiple edges, effectively using multi-core processors. Within this parallel section, a loop (indicated by 'OMP("omp for")') iterates over all the edges in the graph. During each iteration, the function identifies the vertices (denoted as 'i' and 'j') that are connected by the current edge. A key feature within this loop is the use of in-order prefetching command. This command instructs the CPU to proactively load the data related to an edge positioned `PREFETCH_DISTANCE` (value set to **32** for experimentation) steps ahead of the current edge being processed into the cache.

Algorithm 6 Pseudocode for Graph 500 CSR Out-of-order prefetching

Input: Source vertex $srcvtx$
Output: BFS tree bfs_tree_out, maximum vertex max_vtx_out
Initialize bfs_tree_out and max_vtx_out
Allocate and initialize vertex list $vlist$
while termination condition not met **do**
  Determine the range for this iteration
  for all k in the determined range **do**
    Perform out-of-order prefetching for $vlist$ and related $xadj$ entries
    for each neighbor of the current vertex **do**
      if neighbor has not been visited **then**
        Update the BFS tree with the new information
      end if
    end for
  end for
  Update loop control variables
end while
Release allocated resources
return err

Algorithm 7 Pseudocode for Graph 500 OMP-CSR In-Order Prefetching (gather_edge function)

Input: Edge list IJ, number of edges $nedge$
Procedure: GATHEREDGES
Perform in parallel (OMP):
for $k = 0$ to $nedge - 1$ **do**
  Out-of-order prefetch upcoming entries in IJ if within bounds
  $i \leftarrow$ source vertex of edge k
  $j \leftarrow$ destination vertex of edge k
  if $i \geq 0$ AND $j \geq 0$ AND $i \neq j$ **then**
    Scatter edge i to j and j to i
  end if
end for
Pack edges for efficient access

Similarly, As described in Algorithm 8 the `make_bfs_tree` function, there is a focus on the in-order prefetching within the BFS (Breadth-First Search) main loop. Here, the prefetching command, is designed to load data related to upcoming vertices into the CPU cache in advance of their processing. This prefetching is strategically placed to anticipate the data requirements of future loop iterations, thereby aiming to enhance memory access efficiency and overall performance of the BFS algorithm.

Out of order prefetching - As described in Algorithm 9 The out-of-order prefetching in `make_bfs_tree` function, prefetching **&xadj[XOFF(vlist[k + PREFETCH_DISTANCE])]** , where the prefetch address is dependent on the value within `vlist`, leading to potentially irregular memory access.

Algorithm 8 Pseudocode for Graph 500 OMP-CSR In-Order Prefetching (make_bfs_tree function)

Require: Adjacency list representation of the graph, source vertex $srcvtx$
Ensure: BFS tree bfs_tree_out, maximum vertex max_vtx_out
Initialize BFS tree bfs_tree_out and maximum vertex max_vtx_out
Initialize shared variables $k1$ and $k2$
Perform in parallel (OMP):
while $k1 \neq k2$ **do**
  for $k = k1$ to $oldk2 - 1$ **do**
    In-order prefetch for next iterations if within bounds
    Expand BFS tree using BFS logic (details omitted)
  end for
  Synchronize all threads
  Update shared $k1$ and $k2$ (details omitted)
end while
Clean up resources (details omitted)
return err

Algorithm 9 Pseudocode for Graph 500 OMP-CSR Out-of-Order Prefetching (make_bfs_tree function)

Input: Graph represented by adjacency list $xadj$, source vertex $srcvtx$
Output: BFS tree bfs_tree_out, maximum vertex max_vtx_out
Procedure: MAKEBFSTREE
Initialize bfs_tree_out and max_vtx_out
Initialize shared variables $k1$, $k2$
while $k1 \neq k2$ **do**
  for all k from $k1$ to $oldk2 - 1$ **do**
    Prefetch $vlist[k + \text{PREFETCH\_DISTANCE}]$
    Prefetch $xadj[\text{XOFF}(vlist[k + \text{PREFETCH\_DISTANCE}])]$
    Expand BFS tree (details omitted for brevity)
  end for
  Update $k1$, $k2$ (details omitted for brevity)
end while
Clean up resources (details omitted for brevity)
return err

4) **NAS-Integer Sort** - As described in Algorithm 10, the loop iterates over a set number of keys (NUM_KEYS), which is a fundamental operation in integer sorting algorithms. The `work_buff[key_buff_ptr2[i]]++` operation suggests that the algorithm is counting or classifying keys, a common step in integer sorting algorithms, particularly those based on bucket or counting sort techniques. This prefetching instruction is used to load data from the `key_buff2` array into the CPU cache in advance. The conditional prefetching for `key_buff1` based on `key_buff2` indicates an optimization to handle non-sequential access patterns.

Algorithm 10 Manual Prefetch Injection in NAS-IS

for $i = 0$ **to** $NUM\_KEYS - 1$ **do**
  Increment work buffer for key i
  if defined STRIDE **then**
    Prefetch key_buff2 entry at distance FETCHDIST
  end if
  if $i +$ (FETCHDIST >> 1) $< NUM\_KEYS$ **then**
    Prefetch key_buff1 entry at offset given by key_buff2
  end if
end for

5) **NAS-Conjugate Gradient** - As described in Algorithm 11, In this NAS-Conjugate Gradient workload, the software prefetching is implemented to optimize the matrix-vector multiplication, a critical operation in many scientific computations. The algorithm iterates over the matrix rows, computing the dot product of a matrix row and a vector.
Prefetching Column Indices: This step is where the algorithm prefetches column indices that are `FETCHDIST` steps ahead in the iteration. This prefetching aims to ensure that the elements of the `colidx` array are available in the CPU cache when they are needed, reducing memory access latency.
Prefetching Vector Elements: In this step, the algorithm prefetches elements of the vector p. The prefetching is done slightly ahead of the current iteration point (by half the `FETCHDIST`). This strategy is effective when the access pattern to vector p is not strictly sequential and helps to ensure that the needed data is in the cache when accessed.

Algorithm 11 Manual Prefetch Injection in NAS-CG

Input: Matrix a, vector p, row indices $rowstr$, column indices $colidx$
Output: Resultant vectors q and w
Procedure: MATRIXVECTORMULTIPLY
Perform parallel loop over each row j
for each row j from 1 to (lastrow $-$ firstrow $+ 1$) **do**
  Initialize sum to 0.0
  for each element k in row j **do**
    Add product of $a[k]$ and $p[colidx[k]]$ to sum
    Prefetch Column Indices: Prefetch future column indices at distance FETCHDIST
    Prefetch Vector Elements: Prefetch elements of vector p slightly ahead of the current point
  end for
  Assign sum to $w[j]$ and $q[j]$
end for

6) **Camel - Multi Hashing**
As described in Algorithm 12, the algorithm efficiently manages memory access and computational tasks, primarily using a hash function for indirect data handling and conditional prefetching for cache optimization. It

initializes two arrays, where one stores direct values and the other holds pointers to the first array's elements determined by hashed indices. The core computation iterates over these arrays, optionally employing prefetching based on compile-time flags to enhance cache performance by preloading data ahead of its actual use. This conditional prefetching is carefully controlled to avoid out-of-bounds memory access. The algorithm's effectiveness is evaluated by measuring the execution time, showcasing the impact of prefetching on overall performance. The use of hashing at multiple levels adds computational complexity and demonstrates the algorithm's adaptability to varying hashing intensities.

Algorithm 12 Manual Prefetch Injection in Camel Multi-Hashing

Require: Maximum key size defined as MAX_KEY
Ensure: Efficient memory access with optional prefetching

Initialize two arrays: *array1* and *array2* with size MAX_KEY
for $x = 0$ to $MAX\_KEY - 1$ **do**
  Assign x to *array1[x]*
  Assign address of *array1[hash(x)]* to *array2[x]*
end for
Initialize *sum* to 0
Record start time
for $x = 0$ to $MAX\_KEY - 1$ **do**
  if Prefetching is enabled **then**
    if x is within prefetching range **then**
      Prefetch data from *array2* at offset positions
    end if
  end if
  Add hashed values of elements in *array2[x]* to *sum*
end for
Calculate elapsed time
Output the result and time taken

C. Automated Software Prefetching Algorithm

The algorithm implemented in this study is based on the foundational work presented by Sam Aimsworth [1]. We have made significant structural modifications to enhance the overall functionality, efficiency, reliability and readability of the code. These changes are explained in detail under the section IV-D. These adaptations were necessary to align the algorithm more closely with the specific objectives and constraints of our research to test using newer and sparser versions of the workloads.

As shown in Listing 1, the C++ code demonstrates a high level overview of our code for generating software prefetches automatically. This is implemented as a LLVM IR pass which is used within Clang.

The functionality of each function in the automated software prefetching system is described below:

- **DepthFirstSearch Function (Lines 1-20)**: A recursive function to identify induction variables in the source operands of an instruction. It maintains a dictionary to track potential induction variables.
- **GeneratePrefetches Function (Lines 21-31)**: Identifies prefetch targets within loops by iterating over loads and applying `DepthFirstSearch`.
- **filter_out_unsuitable_prefetches Function (Lines 32-37)**: Removes prefetch targets that are unsuitable due to function calls, potential faults, or complex phi nodes.
- **emit_prefetch_and_address_code Function (Lines 38-45)**: Processes each prefetch target, calculating offsets and updating instructions.
- **copy_and_update_instructions Function (Lines 46-53)**: Copies the instruction set and updates each instruction based on induction variable use and calculated offset.
- **replace_with_prefetch Function (Lines 54-59)**: Replaces instructions in the copied set with prefetch instructions and places them before the load in the program.

Listing 1. Automated Prefetch Generation Algorithm

```
1  def DepthFirstSearch(instruction):
2      ind_var_candidates = {}
3
4      for operand in instruction.source_operands:
5          if is_induction_variable(operand):
6              ind_var_candidates[operand] = {
                   ↪ instruction}
7          elif is_loop_variable(operand):
8              ivar, ivar_set = DepthFirstSearch(
                   ↪ loop_def(operand))
9              if ivar is not None:
10                 ind_var_candidates[ivar] = {
                       ↪ instruction}.union(
                       ↪ ivar_set)
11
12     if len(ind_var_candidates) == 0:
13         return None
14     elif len(ind_var_candidates) == 1:
15         return next(iter(ind_var_candidates.items
               ↪ ()))
16
17     closest_ivar =
           ↪ find_closest_loop_induction_var(
           ↪ ind_var_candidates)
18     return merge_instructions_based_on_indvar(
           ↪ closest_ivar, ind_var_candidates)
19
20 def GeneratePrefetches():
21     prefetch_targets = {}
22     for load in loop_loads():
23         ind_var, ivar_set = DepthFirstSearch(load
               ↪ )
24         if ind_var is not None:
25             prefetch_targets[load] = (ind_var,
                   ↪ ivar_set)
26
27     filter_out_unsuitable_prefetches(
           ↪ prefetch_targets)
28     return emit_prefetch_and_address_code(
           ↪ prefetch_targets)
29
30 def filter_out_unsuitable_prefetches(
       ↪ prefetch_targets):
31     for target in list(prefetch_targets):
32         if has_function_calls(target) or
               ↪ causes_faults(target) or
               ↪ has_complex_phi_nodes(target):
33             del prefetch_targets[target]
34
```

```
def emit_prefetch_and_address_code(
    ↪ prefetch_targets):
    for load, (ind_var, instruction_set) in
        ↪ prefetch_targets.items():
        offset = calculate_offset(load, ind_var)
        updated_instructions =
            ↪ copy_and_update_instructions(
            ↪ instruction_set, ind_var, offset)
        replace_with_prefetch(
            ↪ updated_instructions, load)

def copy_and_update_instructions(instruction_set,
    ↪  ind_var, offset):
    copied_instructions = copy_instructions(
        ↪ instruction_set)
    for inst in copied_instructions:
        if uses_induction_var(inst, ind_var):
            update_instruction(inst, ind_var,
                ↪ offset)
    return copied_instructions

def replace_with_prefetch(instructions, load):
    for inst in instructions:
        if is_copy_of(inst, load):
            convert_to_prefetch(inst)
    place_instructions_before_load(load,
        ↪ instructions)
```

D. Low level code overview

Enhancements and added functionality were done to the following functions which are wrapped around a top level **SwPrefetchPass** which is a custom LLVM Function pass. This pass is a transformation and analysis that runs over the source code of the benchmarks. There were a lot of modifications but we will focus on highlighting a few important ones. All low level implementation of the below modifications are shown in (Listings 2, 3, and 4)

- **Early Loop Termination**: Introduced an efficiency check to return early if the instruction is not part of a loop, avoiding unnecessary processing for non-relevant instructions.
- **Simplified Load Instruction Handling**: Employed standard C++ algorithms for handling load instructions, enhancing both clarity and efficiency.
- **New Function *updateRoundInstsAndPhi***: Centralized the logic for phi node and round instruction updates into a separate function for improved maintainability and readability.
- **Refined Phi Node Comparison**: Simplified the phi variable comparison and update logic for better readability and reduced error potential.
- **Null Pointer Checks**: Implemented explicit checks for null pointers to increase the function's safety and robustness.
- **Hoisting Eligibility Flag**: Added a flag to assess an instruction's eligibility for hoisting based on loop-invariance, simplifying the control flow.
- **Consolidated Hoisting Logic**: Unified the hoisting logic within the function to enhance readability and maintainability.
- **Early Null Checks**: Introduced early return conditions for null pointer checks for enhanced safety.
- **Refactor Backedge Checks**: Implemented *isValidBackedge*, a helper function to validate backedges, thereby improving code clarity.
- **Streamlined Loop over PHINodes**: Optimized the PHINode iteration process, making the code more concise.
- **Helper Function for GEP Induction Variable Validation**: Added *isValidGEPInductionVariable* to validate `GetElementPtrInst` as an induction variable.
- **Simplified Backedge Determination**: Refined the process of determining and validating backedges for better readability.
- **Loop Restructuring**: Restructured loops for greater clarity and efficiency, isolating each loop's functionality into separate helper functions.
- **Introduction of *processLoadInst* and *processLoads* Functions**: Developed new functions to handle load instruction processing and to manage identified loads for prefetching, contributing significantly to modularizing the codebase.

Listing 2. Enhanced Depth First Search Function

```
bool depthFirstSearch (Instruction* I, LoopInfo &
    ↪ LI, Instruction* &Phi, SmallVector<
    ↪ Instruction*,8> &Instrs, SmallVector<
    ↪ Instruction*,4> &Loads, SmallVector<
    ↪ Instruction*,4> &Phis, std::vector<
    ↪ SmallVector<Instruction*,8>>& Insts)
{
    Use* u = I->getOperandList();
    Use* end = u + I->getNumOperands();
    SetVector<Instruction*> roundInsts;
    bool found = false;

    // Enhanced: Early loop termination for non-
        ↪ loop instructions
    if (!LI.getLoopFor(I->getParent())) {
        return false;
    }

    for(Use* v = u; v<end; v++) {
        // Original logic...

        // Enhanced: Simplified logic for load
            ↪ instruction handling
        else if(LoadInst * linst = dyn_cast<
            ↪ LoadInst>(v->get())) {
            auto it = std::find(Loads.begin(),
                ↪ Loads.end(), linst);
            if (it != Loads.end()) {
                int lindex = std::distance(Loads.
                    ↪ begin(), it);
                Instruction* phi = Phis[lindex];
                // Enhanced: Simplified phi
                    ↪ comparison and update
                    ↪ logic
                updateRoundInstsAndPhi(LI, Phi,
                    ↪ phi, roundInsts, Insts[
                    ↪ lindex], found);
            }
        }
        // Continue original logic...
    }

    if(found) for(auto q : roundInsts) Instrs.
        ↪ push_back(q);
    return found;
```

```
}

// Enhanced: New function to simplify phi node comparison and update
void updateRoundInstsAndPhi(LoopInfo &LI, Instruction* &Phi, Instruction* phi, SetVector<Instruction*>& roundInsts, SmallVector<Instruction*,8>& insts, bool& found) {
    if (!Phi || LI.getLoopFor(Phi->getParent())->isLoopInvariant(phi)) {
        roundInsts.clear();
        for(auto q : insts) {
            roundInsts.insert(q);
        }
        Phi = phi;
        found = true;
    } else if (!LI.getLoopFor(phi->getParent())->isLoopInvariant(Phi)) {
        // Do nothing if phi is not older than Phi
    }
}
```

Listing 3. Enhanced GEP-Based Induction Variable Identification

```
GetElementPtrInst *getGEPBasedInductionVar(Loop* L) const {
    // Early null checks for Loop and Header
    if (!L || !L->getHeader()) return nullptr;

    BasicBlock *H = L->getHeader();
    BasicBlock *Backedge = L->getLoopLatch();
    if (!H || !Backedge) return nullptr;

    // Simplified backedge determination and validation
    Backedge = *PI++;
    Incoming = (PI != pred_end(H)) ? *PI++ : nullptr;
    if (!Incoming || PI != pred_end(H) || !L->contains(Backedge)) return nullptr;

    // Streamlined loop over PHINodes
    for (PHINode &PN : H->phis()) {
        GetElementPtrInst *Inc = dyn_cast<GetElementPtrInst>(PN.getIncomingValueForBlock(Backedge));
        if (Inc && isValidGEPInductionVariable(Inc, &PN)) {
            return Inc;
        }
    }
    return nullptr;
}

// Helper function to check backedge validity
bool isValidBackedge(const Loop* L, BasicBlock* Incoming, BasicBlock* Backedge) const {
    // ... Original implementation ...
    return true; // Simplified for illustration
}

// Helper function to validate GEP induction variable
bool isValidGEPInductionVariable(const GetElementPtrInst* Inc, const PHINode* PN) const {
    // ...Original implementation ...
    return false; // Simplified for illustration
}
```

Listing 4. Enhanced runOnFunction Method and Helper Functions

```
bool runOnFunction(Function &F) override {
    LoopInfo &LI = getAnalysis<LoopInfoWrapperPass>().getLoopInfo();
    bool modified = false;

    SmallVector<Instruction*,4> Loads;
    SmallVector<Instruction*,4> Phis;
    SmallVector<int,4> Offsets;
    SmallVector<int,4> MaxOffsets;
    std::vector<SmallVector<Instruction*,8>> Insts;

    // Enhanced: Loop restructuring for clarity and efficiency
    for(auto &BB : F) {
        for (auto &I : BB) {
            if (auto *i = dyn_cast<LoadInst>(&I)) {
                if(LI.getLoopFor(&BB)) {
                    // Enhanced: Code refactoring for depth-first search
                    processLoadInst(i, LI, Loads, Phis, Insts, modified);
                }
            }
        }
    }

    // Enhanced: Refactored loop for processing Loads
    processLoads(LI, Loads, Phis, Insts, modified);

    return modified;
}

void processLoadInst(LoadInst *i, LoopInfo &LI, SmallVector<Instruction*,4> &Loads, SmallVector<Instruction*,4> &Phis, std::vector<SmallVector<Instruction*,8>> &Insts, bool &modified) {
    // Function implementation...
}

void processLoads(LoopInfo &LI, SmallVector<Instruction*,4> &Loads, SmallVector<Instruction*,4> &Phis, std::vector<SmallVector<Instruction*,8>> &Insts, bool &modified) {
    // Function implementation...
}
```

V. Our Testing Results

A. System Configuration

Table I shows the system specifications where we conducted our experiments with the sparse workloads. The Operating System running is **Ubuntu 22.04**.

B. Result Evaluation

Implementing our manual prefetch and automated prefetch methodology discussed in the previous sections, Section IV-B and Section IV-C, we have generated our set of results on the benchmarks we had discussed. All set of results contain the execution times of the automatic prefetching algorithm, the manual prefetching algorithms (Manual In-order and Manual Out-of-order in the case of Graph500) and no-prefetching implemented.

TABLE I
SYSTEM SPECIFICATIONS

Component	Specification
CPU	Intel Core i7-9750 CPU @ 2.60 GHz
Memory	16GB RAM
Architecture	x86_64
Cores	6
L1D	192 KiB
L1i	192 KiB
L2	1.5 MiB
L3	12 MiB

Table II contains our comprehensive results. For the NAS-CG and NAS-IS workloads, we highlight the array size being used for testing, for scale. Also, for all the Graph500 workloads, we have included the Harmonic mean of the Traversed Edges Per Second (TEPS), which is a signature metric for measuring TEPS and change in TEPS over benchmarks. It is a useful metric to measuring comparatively in between different number of iterations, but for the purposes of our project, we are keeping number of iterations constant.

Following is the breakdown of the results by benchmark:

1) **NAS-IS:** Both Manual and Auto prefetching provide a significant speedup over no prefetching implementation. Although manual prefetching provides marginally better results than auto, auto is a better alternative for significantly less workload on the programmer, for very competitive results.
2) **NAS-CG:** Automatic prefetching has good speedup over no prefetching implementation. However, manual prefetching shows significant slowdown compared to no prefetching. Manual prefetching requires a deep understanding of the specific access patterns which can be challenging for complex algorithms like CG. This has occured due to inaccurate prefetching decisions. If the developer insights are not precise, manual prefetching can lead to fetching data that is not needed soon, resulting in waster memory bandwidth and cache space.
3) **Graph500 - Sequential Compressed-Sparse-Row:** Manual prefetching shows a clear advantage over auto prefetching, especially out-of-order. The harmonic mean TEPS also follows the same trend, having lowest value at no prefetching, then higher at auto prefetching and highest value at manual out-of-order. The manual injections prove to be more helpful for prefetching. Since we have a control over prefetch distance in manual prefetching, the performance is increased due to the fine tune to match the access pattern of the CSR format in Graph500.
4) **Graph500 - OpenMP Compressed-Sparse-Row:** Auto and Manual out-of-order execution are very close, manual being marginally faster, while both approaches give a significantly better execution time over no prefetching.
5) **Graph500 - Sequential List:** Auto prefetching is marginally better than no prefetching, while both the manual prefetching versions have longer execution times. The intuitive reason for that is because of the underlying data structure being implemented for this workload. Sequential list does not significantly benefit from prefetching, but will show massive improvements from parallelization. The intuition for choosing a list-based graph implementation is mentioned in Subsection IV-A(Under the workload's **Use Case**):
6) **Camel - Multi-hashing memory access:** Both manual and auto prefetching gives significant speedup over no prefetching, with manual having slightly better performance than auto-prefetching.

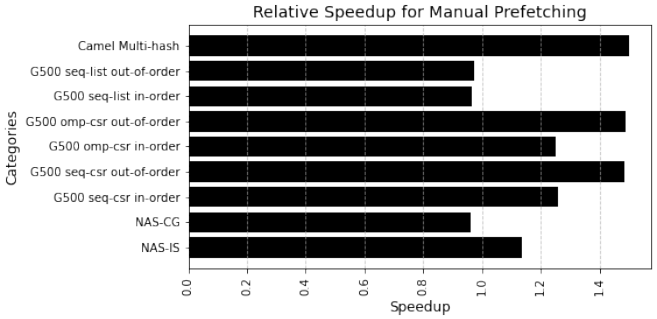

Fig. 1. Speedup of Manual Prefetching over No Prefetching

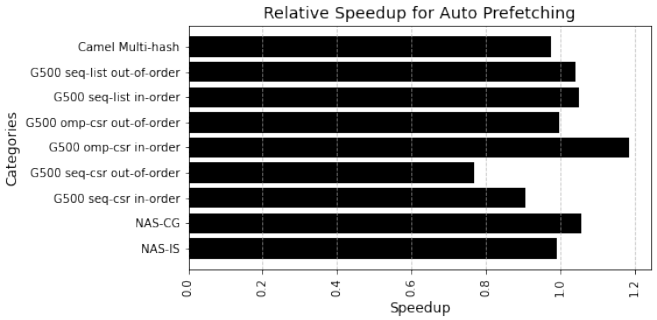

Fig. 2. Speedup of Automatic Prefetching over No Prefetching

Figures 1, 2 and 3 all present the relative speedups. Figure 1 presents relative speedup of manual prefetching over no

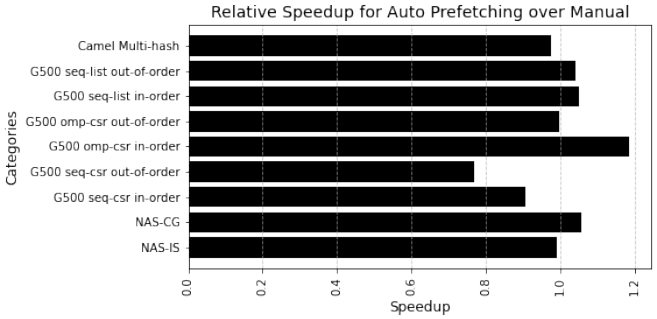

Fig. 3. Speedup of Automatic Prefetching over Manual Prefetching

TABLE II
COMPILED NUMERICAL RESULTS FOR PREFETCHING WORKLOADS

NAS-IS			
Prefetch type	***Iterations***	***Array Size***	***Mean Execution Time (s)***
Auto	10	33554432	1.12
Manual	10	33554432	1.11
None	10	33554432	1.26
NAS-CG			
Prefetch type	***Iterations***	***Array Size***	***Mean Execution Time (s)***
Auto	75	150000	22.17
Manual	75	150000	23.38
None	75	150000	22.45
Graph500 - Sequential Compressed-Sparse-Row			
Prefetch type	***Iterations***	***Harmonic mean TEPS***	***Mean Execution time (s)***
Auto	64	135453625.59	0.12386
Manual in-order	64	149765871.47	0.11202
Manual out-of-order	64	176750805.56	0.09492
None	64	119086727.03	0.14088
Graph500 - OpenMP Compressed-Sparse-Row			
Prefetch type	***Iterations***	***Harmonic mean TEPS***	***Mean Execution time (s)***
Auto	64	8090654.54	0.20736
Manual in-order	64	6830231.72	0.24562
Manual out-of-order	64	8.127981.68	0.206411
None	64	7621762.73	0.30728
Graph500 - Sequential List			
Prefetch type	***Iterations***	***Harmonic mean TEPS***	***Mean Execution time (s)***
Auto	64	591598.17	2.83588
Manual in-order	64	563949.24	2.97491
Manual out-of-order	64	569621.92	2.94529
None	64	584738.01	2.86915
Camel - Multi-hashing memory access			
Prefetch type	***N/A***	***Array Size***	***Mean Execution Time (s)***
Auto	N/A	33554432	5.537
Manual	N/A	33554432	5.396
None	N/A	33554432	8.100

prefetching. Based on the results, most workloads see a speedup of about **1.2x**, besides Graph500 Sequential List and NAS-CG. The reasons for so are explained in the points above. Figure 2 shows the relative speedups for automated prefetching algorithm with respect to no prefetching. All the algorithms show closely related execution times, hence not showing significant speedups. However, for Graph500 sequential compressed sparse row shows a very significant speedup.

The main discussion point is the relative speedup of auto prefetching over manual prefetching, presented in figure 3. The significant improvement of our auto-fetch algorithm can be seen in the Graph500 Sequential Compressed Sparse Row in-order, showing over **1.1x** speedup, while NAS-CG and Graph500 Sequential List also show slight speedups. Many other workloads shows either no speedup, or slowdowns which might be a factor of our algorithm's implementation with those workloads, or the nature of the execution of the workloads with prefetching algorithms.

In conclusion, there is a lot of benefit in using prefetching algorithms since there is significant benefit in implementing them on sparse workloads. Especially, if there is an automated prefetching algorithm in place, it can boost execution time for many custom workloads. The tremendous speedup improvement due to the auto-prefetching algorithm, which also includes the time saved and the manual labor saved for the programmer, is a big enough performance boost.

VI. Conclusion

In our study, we implemented a custom version of an automated software based prefetching algorithm tested on multiple sparse workloads tailored for indirect memory acesses. We tested against multiple sparse benchmarks, using their updated source code and tested the performance of manual prefetching against them. We have also implemented a compiler-based automatic prefetching algorithm, explained its implementation and compared its performance with manual prefetching and no prefetching counterparts, and provided comprehensive results for the same. We have concluded that our automatic prefetching algorithm provides good results comparatively. Also we have researched some other advancements made in different directions in different studies and discussed how we can implement their work alongside our algorithm to improve performance. This is discussed in the upcoming section VII.

VII. Discussion

Few points of improvement for our research methodology. This can be extended as future scope.

A. A novel approach of combining Software Prefetching with Cache Line Mechanism

From the paper **IMP: indirect memory prefetcher** [2], the authors have implemented a novel **Partial Cacheline Ac-**

cessing Mechanism. This mechanism is particularly beneficial in the context of sparse data structure, where not all data in cacheline is relevant or needed. By selectively fetching only the necessary parts of the block, this mechanism optimizes data retrieval, further bolstering the system's efficiency in handling sparse data structures. Partial cacheline accessing is enabled in both NoC (Network-on-Chip) and DRAM (Dynamic Random Access Memory) which presents a promising approach to optimizing memory operations for sparse data workloads when integrated with automated software prefetching.

Motivation:

Automated software prefetching aims to anticipate future memory access and proactively bring data into cache before it is needed. This can significantly improve performance and reduce manual effort and time as we have seen in our experiments. However, traditional prefetching approahces typically fetch entire cachelines, which can lead to cache pollution and waster bandwidth for sparse workloads where only a small portion of the data is acutally needed.

Partial cacheline accessing mechanism (PCAM) can be a promising solution to this problem by allowing the selective fetching of only the necessary part of a cacheline. This can be benefical for sparse workloads by:

- **Reducing cache pollution**: By fetching only the relevant data, PCAM prevents unnecessary filling of the cache with unused inforamtion, leaving more space for frequently accessed data.
- **Improving memory bandwidth utilization**: By fetching smaller data chunks, PCAM reduces the amount of data that needs to be transferred through the memory hierarchy, leading to better overall bandwidth utilization.
- **Decreasing memory access latencies** : By fetching only the required data, PCAM can potentially reduce the latency of accessing the desired information, especially for data located farther away in the memory hierarchy.

Advantages:

We hypothesize the potential advantages by combining automated software prefetching with PCAM.

- **Improved Prefetching Accuracy**: Software prefetching can predict future memory accesses based on program analysis and runtime information. By incorporating PCAM, the prefetching instructions can be tailored to fetch only the necessary portions of the data, leading to more accurate predictions and improved performance.
- **Reducing Prefetching Overhead**: With PCAM, the prefetched data can be precisely adjusted to the actual needs, reducing overhead and improving efficiency.
- **Increased Efficiency for Sparse Workloads**: As we know, sparse workloads often exhibit irregular memory access patterns which makes them particularly susceptible to cache pollution and wasted bandwidth. PCAm can significantly improve the perforamnce of such workloads by prefetching only the relevant data, leading to a better cache utilization.

Disadvantages: We identify a set of potential disadvantages or challenges associated with combining these techniques

- **Implementation Complexity**: Integrating PCAM with software prefetching requires additional hardware or software support, which can increase the overall system complexity and development effort.
- **Potential Performance Overhead**: The additional processing required for handling partial cacheline accesses may introduce some performance overhead, especially on simple hardware platforms.
- **Storage Cost of Partial Cacheline Accessing**: As mentioned in the paper [2], there is a storage cost due to having sector caches in L1 and L2 to support partial cacheline accessing. In the paper, each L1 cacheline is split into 8 sectors, and each L2 cacheline into 2 sectors. This can introduce storage overhead for L1 and L2.

B. Voyager: 'A hierarchical neural model of data prefetching' [12]

From the paper 'A hierarchical neural model of data prefetching' by Shi et al. [12], the Voyager is one of the novel neural network implementations for boosting software prefetching usage in all sorts of custom workloads. The model learns data as well as address correlations between irregular memory accesses and has been shown to provide significantly better results on certain workloads [12].

Motivation: There are certain limitations that automated software prefetching will face that cannot be overcome. Using Machine Learning based approaches for predicting prefetch instructions and detecting irregular access patterns, using them for further boosting targeted prefetching.

Potential implementation Methodology:

1) If a pre-trained model is available, using it during compile time alongside our software auto-prefetching algorithm to generate targeted prefetch instructions for specific locations would improve the efficacy of the algorithm embedding the prefetching instructions.
2) If there are no pre-trained models available, or none of the available ones are suitable for some custom workload being used, then generates memory access traces, and training a new model would consume some overhead time. But once a sufficiently big model is available, the same approach as above can be used.
3) Results generated from each prefetching run can also be used as training data in a reinforcement-learning fashion in order to improve existing pre-trained models.

Advantages: The main advantage is that we get to use an extremely complex learning algorithm to find prefetching instruction based on existing patterns, which cannot be generated by any in-place functions. Also, pre-trained models can be reused in similar workloads, making the overhead of using this approach much smaller.

Disadvantages: The training overhead of this algorithm for each new type of workload will be huge, especially if there is not much memory-access data readily available. Existing

model design will also need to be trained on GPU/TPU resources in order to generate usable models that give significant results.

C. Advanced Benchmark analysis

This study relied on the latest versions of sparse workloads such as NAS benchmarks, and Graph500. Implementing and testing new and more complex benchmarks, such as PageRank, Database operations, Monte Carlo simulations, etc might provide interesting results. In addition to this, custom workloads focused more directly on indirect memory accesses within sparse workloads would be ideal. Even implementing new, more targeted benchmark suites like HPCG (High Performance Conjugate Gradient) and testing performance/relative speedups of prefetching algorithms on HPC systems could prove worthwhile [13].

ACKNOWLEDGMENT

We would like to thank Dr. Venkatesh Akella and Jaxon Brown since this study was a product of the final term paper for EEC 270: Computer Architecture. Their support and direction throughout the process of this study was vital for getting it in the direction it has come.

REFERENCES

[1] Sam Ainsworth and Timothy M. Jones. 2019. Software Prefetching for Indirect Memory Accesses: A Microarchitectural Perspective. ACM Trans. Comput. Syst. 36, 3, Article 8 (August 2018), 34 pages. https://doi.org/10.1145/3319393

[2] X. Yu, C. J. Hughes, N. Satish and S. Devadas, "IMP: Indirect memory prefetcher," 2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), Waikiki, HI, USA, 2015, pp. 178-190, doi: 10.1145/2830772.2830807.

[3] Mustafa Cavus, Resit Sendag, and Joshua J. Yi. 2020. Informed Prefetching for Indirect Memory Accesses. ACM Trans. Archit. Code Optim. 17, 1, Article 4 (March 2020), 29 pages. https://doi.org/10.1145/3374216

[4] APT-GET: Profile- Guided Timely Software Prefetching. Saba Jamilan, Tanvir Ahmed Khan, Grant Ayers, Baris Kasikci, and Heiner Litz. 2022. ACM

[5] D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. In ASPLOS, 1991.

[6] Benchmarks, NAS Parallel. "NAS Parallel Benchmarks." CG and IS (2006).

[7] T. Suzumura, K. Ueno, H. Sato, K. Fujisawa and S. Matsuoka, "Performance characteristics of Graph500 on large-scale distributed environment," 2011 IEEE International Symposium on Workload Characterization (IISWC), Austin, TX, USA, 2011, pp. 149-158, doi: 10.1109/I-ISWC.2011.6114175.

[8] T. C. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In ASPLOS, 1992.

[9] Jaekyu Lee, Hyesoon Kim, and Richard Vuduc. 2012. When prefetching works, when it doesn't, and why. ACM Transactions on Architec- ture and Code Optimization (TACO) 9, 1 (2012), 1–29.

[10] Leskovec, Jure, et al. "Kronecker graphs: an approach to modeling networks." Journal of Machine Learning Research 11.2 (2010).

[11] L. Dagum and R. Menon, "OpenMP: an industry standard API for shared-memory programming," in IEEE Computational Science and Engineering, vol. 5, no. 1, pp. 46-55, Jan.-March 1998, doi: 10.1109/99.660313.

[12] Zhan Shi, Akanksha Jain, Kevin Swersky, Milad Hashemi, Parthasarathy Ranganathan, and Calvin Lin. 2021. A hierarchical neural model of data prefetching. In Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21). Association for Computing Machinery, New York, NY, USA, 861–873. https://doi.org/10.1145/3445814.3446752

[13] HPCG Benchmark: https://www.hpcg-benchmark.org

[14] Grant Ayers, Heiner Litz, Christos Kozyrakis, and Parthasarathy Ranganathan. 2020. Classifying Memory Access Patterns for Prefetching. In Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20). Association for Computing Machinery, New York, NY, USA, 513–526. https://doi.org/10.1145/3373376.3378498