Magellan: A High-Performance Loop-Guided Prefetcher for Indirect Memory Access

Gelin Fu
Xi'an Jiaotong University
Xi'an, China
fugelin@stu.xjtu.edu.cn

Tian Xia
Xi'an Jiaotong University
Xi'an, China
tian_xia@xjtu.edu.cn

Mingzhuo Yin
Xi'an Jiaotong University
Xi'an, China
ymzymz@stu.xjtu.edu.cn

Prashant J. Nair
The University of British Columbia
Vancouver, Canada
prashantnair@ece.ubc.ca

Mieszko Lis
The University of British Columbia
Vancouver, Canada
mieszko@ece.ubc.ca

Pengju Ren
Xi'an Jiaotong University
Xi'an, China
pengjuren@xjtu.edu.cn

Abstract

Graph analytics and sparse linear algebra applications heavily rely on indirect memory access (IMA). IMAs are characterized by poor temporal and spatial locality, which causes frequent high-latency DRAM accesses. While dedicated hardware prefetchers for IMA have been explored, they target narrow access patterns and tend to introduce significant hardware complexity. Software prefetching offers a promising alternative, leveraging compiler analysis to prefetch indirection patterns. However, existing software prefetchers struggle with sparse applications due to limited loop iterations and complex IMA patterns across nested loops.

We propose *Magellan*, a novel loop-guided software prefetcher designed to detect and schedule IMA prefetches efficiently. Magellan introduces two key innovations: (1) extracting dependence graphs across loop levels to detect complex IMA patterns and (2) capturing inner-outer loop semantics to prefetch for both current and future iterations. We evaluate Magellan on 14 memory-intensive benchmarks using real-world datasets from social networks and web graphs. Compared to the best existing IMA software prefetcher, Magellan reduces cache misses by 25% and dynamic instruction counts by 14% on average. This results in a 1.14× average speedup, with performance gains of up to 1.41×.

CCS Concepts

• **Software and its engineering → Compilers**.

Keywords

Software Prefetcher, Indirect Memory Access, Memory Hierarchy

ISCA '25, Tokyo, Japan

ACM ISBN 979-8-4007-1261-6/25/06
https://doi.org/10.1145/3695053.3731054

ACM Reference Format:
Gelin Fu, Tian Xia, Mingzhuo Yin, Prashant J. Nair, Mieszko Lis, and Pengju Ren. 2025. Magellan: A High-Performance Loop-Guided Prefetcher for Indirect Memory Access. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA '25), June 21–25, 2025, Tokyo, Japan.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3695053.3731054

1 Introduction

Sparse irregular algorithms are key to various application domains, including graph analysis [50, 53], high-performance computing [43], and machine learning [20]. These algorithms often rely on indirect memory accesses (IMAs), where elements in one array are used as indices to access another array. IMAs, however, introduce significant challenges due to their highly irregular index values. This irregularity results in frequent cache misses and high-latency DRAM accesses, creating a bottleneck that severely impacts performance. This paper focuses on developing efficient prefetching strategies to alleviate the performance overhead of IMAs.

Memory access prefetching is a well-established technique for mitigating latency bottlenecks and can be implemented in hardware or software [27]. Several studies have proposed specialized *hardware prefetchers* tailored to IMA patterns [2, 3, 22, 23, 28, 29, 59, 60, 68, 75, 82, 84]. For example, IMP [84] and DMP [29] exploit arithmetic regularity in memory address histories to prefetch indirection patterns. Event-Trigger [3] and Prodigy [75] annotate indirection semantics in software, which are then offloaded to programmable hardware at runtime to issue prefetching requests. Although these prefetchers deliver substantial performance improvements, they are impractical for real-world scenarios due to their reliance on narrow access patterns and the additional chip area required for custom logic. Consequently, commercial CPUs, such as Intel Xeon [36], use simpler prefetchers like next-line [73] and stride prefetchers [21], which cannot effectively handle IMA patterns.

Why Use Software Prefetchers: Compiler-based software prefetching techniques have been proposed to overcome the limitations of hardware prefetchers. Software prefetchers can respond to dynamic program behaviors without increasing processor complexity. Most existing software prefetching approaches focus on regular access patterns, such as stride accesses [35, 54] and linked-list traversals [19, 49, 71, 80]. Some software prefetchers like *SW Prefetch* [4] aim to specifically target IMA patterns of the form *x[a[i]]*. They identify load instructions referencing loop induction variables and

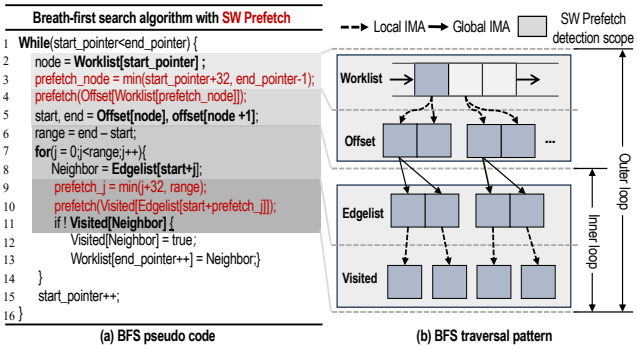

Figure 1: Pseudo code (a) and traversal pattern (b) of breath-first-search (BFS) graph algorithm from GAP benchmark [13] with SW Prefetch [4]. In 85.3% of cases, indices like j+32 exceed the inner-loop boundary and are replaced with the boundary value – thus negating any software prefetch.

use a depth-first search algorithm to generate prefetching instructions. Despite this, our studies observed that *SW Prefetch* struggled to match the performance of advanced hardware prefetchers like DMP [29] (see Fig.18 in Section 5.4). We observed that the effectiveness of such software prefetchers is limited in sparse irregular applications due to: (1) sparse data distributions that restrict prefetching opportunities, and (2) their inability to prefetch complex indirection patterns found in nested loops.

Challenges: The limitations of SW Prefetch can be illustrated using the breadth-first search (BFS) algorithm shown in Fig.1(a) and its indirection pattern in Fig.1(b). SW Prefetch restricts prefetch addresses to the current loop iteration to ensure valid look-ahead prefetches. For example, to prefetch the IMA load *Visited* (line 11), it computes the prefetch index by incrementing the induction variable *j* with an ahead distance and comparing it to the loop boundary *range* (line 9). This index is then used to generate an indirect prefetch for *Visited* (line 10). However, when processing sparse matrices, such as web graphs or road networks [13, 16, 18, 46], most vertices have few neighbors, resulting in tight inner loops. Our experiments show that in 85.3% of cases, indices like *j+32* exceed the loop boundary and are replaced with the boundary value. Thus, most prefetches redundantly target the same address, *Visited[Edgelist[start+range]]*, which significantly reduces their effectiveness. Furthermore, BFS features two types of IMAs that challenge SW Prefetch. **Local IMAs**, such as accesses to *Offset* and *Visited*, occur within the same loop. In contrast, **global IMAs**, such as accesses to *Edgelist*, depend on an outer-loop variable (*start*). High-performance IMA prefetchers require intelligent mechanisms to detect different IMA patterns from the source program.

To quantify the impact of these factors, Fig. 2 presents the cache efficiency of various graph analysis and high-performance computing algorithms using hardware and software prefetchers. Stride and next-line hardware prefetchers eliminate only a small fraction of cache misses. Adding SW Prefetch reduces more cache misses, but over 60% of IMA-related misses remain unresolved. This issue is particularly pronounced in algorithms like BFS, SSSP, and BC, where more than 30% of cache misses stem from global IMAs.

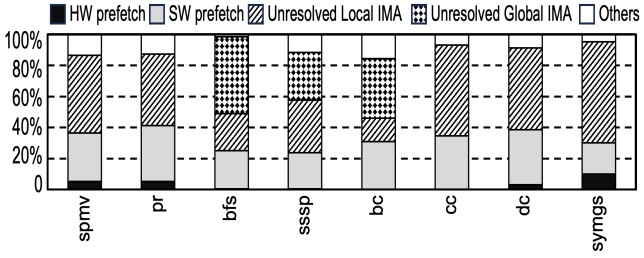

Figure 2: LLC miss breakdown of various sparse applications on Intel Kabylake. On average, 60% of the IMA-related misses remain unresolved despite using prefetchers.

These misses are challenging to prefetch effectively, highlighting the limitations of current software prefetching techniques.

Goal: This work aims to enhance the performance of software prefetchers for sparse irregular applications by addressing three key challenges: (1) exploiting more prefetch opportunities in loops with a small number of iterations; (2) accurately detecting complex IMAs that span nested loops. (3) ensuring that the inserted prefetch instructions do not compromise program correctness. We propose the **Magellan** software prefetcher to address these challenges.

Insights and Mechanisms: Magellan uses the observation that IMAs often operate within nested loops where inner loops are interdependent. To exploit inter-loop prefetch opportunities, Magellan detects the **nested loop pattern**, a framework that describes the behavior of IMA loads across different loop levels. Based on patterns observed in real-world sparse applications, Magellan classifies nested loop patterns into three categories: stream-in, stream-out, and irregular. For each, it applies a tailored prefetching strategy that balances instruction overhead and prefetch candidate exploration to maximize performance in specific scenarios.

Magellan extends its scope beyond single-level loops by leveraging the **Loop Dependence Graph (LDG)** to enhance the detection of nested IMA patterns. The LDG is a directed graph that captures the reference-chasing sequences of induction variables and loads across multiple loop levels. Using the LDG, Magellan iterates through load instructions to independently detect local and global IMAs, enabling more effective and comprehensive prefetching across nested loops. This approach bridges the gap between theoretical opportunities for prefetching and practical implementation, addressing the complexities of irregular access patterns.

Ensuring Safe Prefetches: Magellan modifies the original program's code structure by explicitly inserting prefetch instructions. However, due to the speculative nature of prefetching, the prefetched addresses may be inaccurate and cross the protection domain, potentially compromising program safety. Traditional approaches, such as *Software Fault Isolation*, validate prefetched addresses by adding conditional bound checks for each address [72, 76]. However, these checks often incur significant performance overhead, slowing execution by up to 2× [77]. Magellan takes a different approach to ensure safe prefetches without sacrificing performance. It identifies memory allocation functions associated with the prefetched data

structures and extends their storage space to *guarantee* prefetch address correctness. Since sparse applications typically involve large datasets, Magellan requires only 0.0036% additional memory.

Key Results: We evaluated Magellan using both real processors and the GEM5 simulator [15], comparing it with five state-of-the-art hardware prefetchers (IPCP [66], Berti [61], IMP [84], Event-trigger [3], and DMP [29]) and three software prefetchers (SW Prefetch [4], Intel OneAPI [65], and APT-GET [38]). The benchmarks included 14 real-world applications spanning sparse linear algebra, graph analytics, and databases. Compared to the best software prefetcher, Magellan reduces cache misses by 25% and dynamic instruction counts by 14% on average, achieving a 1.14× average speedup (up to 1.41×). Magellan performs on par with the best hardware IMA prefetcher, DMP, but without the additional hardware overhead.

Contributions: This paper makes the following contributions:

(1) We propose Magellan, a high-performance software prefetcher designed to efficiently detect and prefetch both local and global indirect memory access (IMA) patterns in sparse applications.
(2) We develop mechanisms for constructing global dependence graphs, detecting nested loop patterns, and avoiding program faults. These enable accurate detection of indirection patterns and loop semantics, while ensuring the safety of prefetching requests.
(3) We provide an LLVM pass that automatically identifies indirection patterns and implements the corresponding prefetching strategies into the original programs, enabling seamless integration into existing workflows.
(4) We extensively evaluate Magellan across diverse workloads, demonstrating that it outperforms state-of-the-art prefetchers. We also provide in-depth analysis and discussions on Magellan's performance and effectiveness.

2 Background and Motivation

Indirect memory access (IMA) refers to data-dependent loads that explicitly link separate data structures. IMAs are prevalent in sparse computing domains such as graph analytics [2, 3, 12, 75] and sparse linear algebra [50, 63, 64, 81]. These applications frequently use IMAs to access neighbor vertices in graphs or non-zero elements within sparse matrices. Due to their substantial size, sparse matrices and graphs typically exceed on-chip cache capacities [10, 18], resulting in irregular IMAs and frequent cache misses. Consequently, these misses lead to high-latency DRAM accesses and degrade performance on contemporary processors.

To mitigate these memory-related bottlenecks, this paper introduces specialized prefetching techniques tailored for sparse applications. We define two fundamental types of IMAs commonly found in these workloads, *local* and *global* indirection, and propose a corresponding application model. Additionally, we present three distinct prefetching strategies designed for different usage scenarios, each associated with varying overheads, and thoroughly evaluate their effectiveness.

2.1 Indirection Types and Application Model

An IMA typically involves two data structures: an index array *a[]* and a target array *x[]*. Depending on how the values in the index array are used to fetch elements from the target array, IMAs can

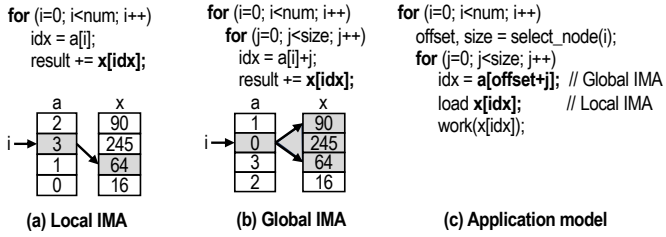

Figure 3: Two basic IMA types: Local IMA (a), global IMA (b), and application model of sparse applications (c).

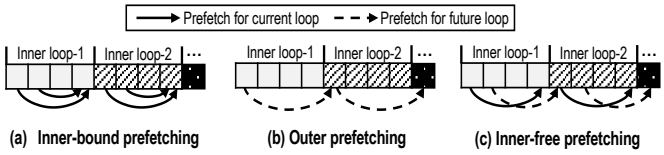

Figure 4: Three prefetching strategies for sparse applications: inner-bound prefetching, inner-free prefetching, and outer prefetching. Inner-free and outer prefetching exploit more prefetch opportunities beyond the current loop.

be categorized as *local* or *global* indirection. Each index value is used in local indirection to fetch a single data element at a time, as shown in Fig.3(a). In contrast, global indirection uses each index as a base address to fetch several consecutive elements, with the offset provided by an inner-loop induction variable, as depicted in Fig.3(b). These local and global IMA types can combine to form complex indirect access patterns.

A common code pattern in many sparse and graph applications involves two-level loops. In the outer loop, a working node is selected at each iteration (e.g., a vertex in a graph, row, or column in a sparse matrix). The inner loop then traverses related elements—such as the edges of a vertex or the non-zero elements of a row or column—with contiguous addresses. Fig. 3(c) provides a code example of this pattern. Here, accessing multiple elements of the selected node (i.e., *idx = a[offset + j]*) represents a global IMA, while loading the property of each element (i.e., *load(x[idx])*) is a local IMA.

2.2 Prefetching Beyond Current Loop

One of the first software techniques targeting IMAs is Ainsworth and Jones's *SW Prefetch* [4]. SW Prefetch primarily focuses on local IMAs within inner loops without considering nested loop structures. Consequently, the address predictions are confined to the current inner loop – a method we refer to as *inner-bound prefetching* (see Fig. 4(a)). However, inner loops are often interrelated through outer loops, presenting additional prefetching opportunities from inter-loop dependencies that SW Prefetch has overlooked.

The first scheme to use nested loop structures is APT-GET [38], which employs *outer prefetching* to prefetch data in outer loops and maintain prefetch timeliness. However, as shown in Fig.4(b), outer prefetching only prefetches data for the next inner loops, which ignores the *intrinsic relationship* between adjacent inner loops and may achieve lower performance due to limited coverage. To exploit more prefetching opportunities, we propose a new prefetching

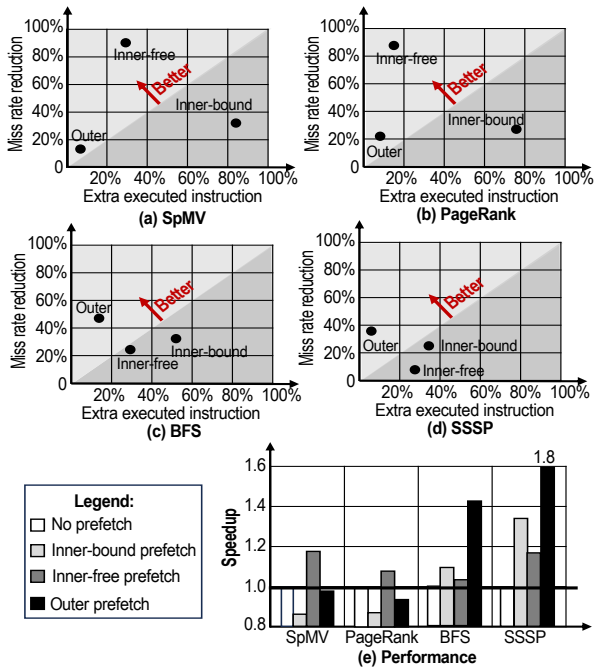

Figure 5: Prefetch effectiveness and performance of IMA prefetch with different prefetching strategies. We evaluate the benefits of reducing the cache miss rate and the overhead of executing extra instructions.

method named *inner-free prefetching*. As shown in Fig.4(c), inner-free prefetching exploits inter-loop opportunities by aggressively issuing prefetches for both current and future inner loops.

However, software prefetching is a complex problem whose effect is influenced by the trade-off between two factors. On the one hand, useful prefetches reduce cache misses. On the other hand, prefetch instructions consume additional CPU cycles. Therefore, a prefetch strategy may result in very different trade-off effects on different applications due to their various loop structures.

To validate our approach, we evaluate the performance of these three prefetching strategies on four sparse applications from the GAP benchmark suite [13]: Sparse Matrix-Vector Multiplication (SpMV), PageRank, Breadth-First Search (BFS), and Single Source Shortest Path (SSSP), using the large *com-LiveJournal* web graph from the SuiteSparse Collection [24]. Figures 5(a)–(d) illustrate the trade-offs of each prefetching strategy, assessing benefits by cache miss reduction rates and overheads by executed instruction counts. We observe that each prefetching strategy exhibits different trade-off tendencies. Specifically, inner-bound prefetching tends to incur higher instruction overhead, while inner-free and outer prefetching have lower instruction overhead, but their effectiveness varies significantly across applications.

As shown in Fig. 5(e), the impact of prefetching strategies differs markedly among applications and can sometimes even degrade performance. Therefore, efficient IMA prefetching necessitates identifying the patterns of nested loop structures and selecting the appropriate prefetching strategy accordingly.

3 The Design of Magellan Prefetcher

Fig. 6 provides an overview of the Magellan prefetcher. It inserts prefetching instructions into a target program through four phases. First, Magellan detects IMA patterns within the target program using a Loop Dependence Graph (LDG), which analyzes dependencies between loops at different levels and their associated loads. Once an IMA is detected, Magellan identifies its nested-loop pattern to describe the relationships between IMAs in the inner and outer loops. Based on this pattern, it implements the appropriate prefetching strategy. Finally, Magellan performs program fault avoidance to ensure that the inserted prefetch instructions are valid and do not cause memory access violations.

3.1 IMA Detection

3.1.1 Loop Dependence Graph (LDG): Abstracting Indirection Across Nested Loops. Sparse applications typically employ two types of indirect memory accesses (IMAs): local and global. Local IMAs occur within a single-level loop, while global IMAs span across nested loops. Detecting these IMAs requires capturing load dependencies between different loop levels – a capability that prior techniques lack. Magellan addresses this challenge using a directed graph called the Loop Dependence Graph (LDG), which abstracts the sequences of load instructions and induction variables in nested loops.

Figure 7 shows a portion of the LDG constructed for the double-nested loop in the BFS algorithm from the GAP benchmark suite [13]. In this graph, each node represents a load instruction or a loop induction variable. A directed edge from node a to node b indicates that node b is data-dependent on node a. A key feature of the LDG is its ability to represent dependencies across different loop levels, providing the offsets for inner-loop global IMAs based on outer-loop variables.

The figure also shows three common structures of global IMAs found in real-world applications, highlighting how cross-loop dependencies are linked in various ways. In Fig. 7(a) and (b), the outer-loop load is directly or indirectly included in the source operand of the global IMA. In contrast, Fig. 7(c) encodes the outer-loop load into the loop iteration condition. During LDG construction, Magellan unifies these forms into a consistent representation to facilitate the detection of global IMAs.

3.1.2 Construction and Traversal of the LDG. Magellan follows the procedure outlined in Algorithm 1 to build a target program's Loop Dependence Graph (LDG). Since local and global IMAs reside within loops, Magellan first searches for basic blocks encapsulated by loops (line 3). Within each block, it leverages insights similar to those used in SW Prefetch [4], recursively iterating over the source operands of each load instruction to construct dependency chains (lines 4 and 12). This backward traversal continues until it encounters another load instruction or an induction variable (line 17). Dependencies related to loop iteration conditions are detected by examining the incoming instructions of the loop (line 7). Using these detected dependencies, Magellan constructs the LDG (line 9).

With the LDG constructed, Magellan begins at each load node and traverses its source operands to detect different types of IMA patterns. If a load depends on another load from the same loop level, it is identified as a local IMA. Conversely, if it depends on an induction variable and another load from an outer loop, it is classified

Figure 6: Magellan takes four steps to insert efficient and safe prefetching instructions for a target program.

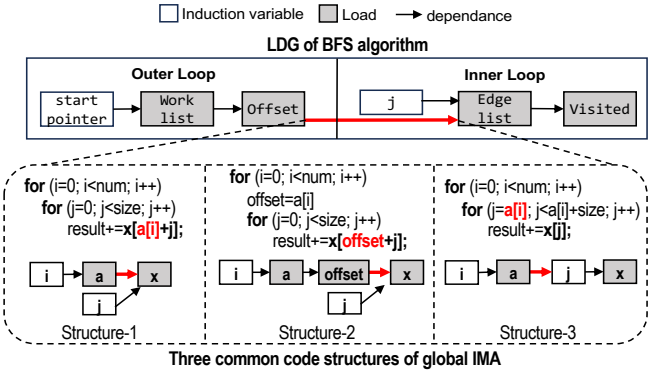

Figure 7: Loop Dependence Graph (LDG) for double nested loop of BFS application and the three common code structures for global IMA.

Algorithm 1 Loop Dependence Graph Construction

```
1:  graph ← {}
2:  for all b in function.basicblocks do
3:      if L = getLoopfor(b) ≠ null then
4:          for all ld in b.loadinsts do
5:              analyze_dfs(ld)
6:          if iv = getInductionVariable(L) ≠ null then
7:              analyze_dfs(getPhiIncoming(iv))
8:          for all p in results.paths do
9:              graph.connect(p)
10: function ANALYZE_DFS(inst)
11:     path ← {}
12:     for all op in inst.operands do
13:         if op is not an instruction then
14:             continue
15:         if op is Call, Store, or Terminator then
16:             continue
17:         if op is LoadInst or InductionVariable then
18:             store path in results
19:         else
20:             store op in path
21:             call analyze_dfs(op)
```

as a global IMA. For instance, in the BFS algorithm, traversing the dependence graph allows Magellan to identify two local IMAs for *Offset* and *Visited*, and one global IMA for *Edgelist*. Having detected all IMAs, Magellan uses the loop structures to implement efficient prefetching strategies.

3.2 Nested-Loop Patterns

3.2.1 Classification. In sparse and graph applications, nested loops frequently handle indirect memory accesses. Based on this observation, we categorize these nested loop structures into three common patterns: stream-in, stream-out, and irregular. Figure 8 shows their respective code structures, memory traversal patterns, and representative use cases. Each pattern incrementally processes elements in the inner loop, aligning with sequential traversals of non-zero elements within matrix rows or edges connected to vertices in graph computations. However, the relationships between iterations of the outer loops vary distinctly across these patterns, as detailed below:

(1) **Stream-in**. In this pattern, the outer loop moves in the same direction as the inner loop, resulting in the inner-loop induction variable j changing continuously across inner-loop iterations. This pattern frequently occurs in sparse linear algebra and graph computations, where rows of matrices or vertices in graphs are processed sequentially. A single-level loop can be seen as a special case of the stream-in pattern, involving only one iteration of the outer loop.

(2) **Stream-out**. In contrast to stream-in, the outer loop moves in the opposite direction to the inner loop, causing the inner-loop induction variable j to exhibit discontinuity between iterations. Typically, this pattern traverses matrix rows or columns in reverse order. For instance, the Symmetric Gauss-Seidel Smoother (SYMGS) in HPCG [25] uses the stream-out pattern during its back-triangular solve, an essential operation in multigrid sparse solvers.

(3) **Irregular**. This pattern features an outer loop whose direction is not fixed, varying dynamically during execution. It commonly appears in graph algorithms, where vertices are accessed in arbitrary sequences at runtime.

3.2.2 Detection. Magellan follows a three-step process to detect the nested loop pattern of an IMA load within nested loops, as shown in Fig. 10. First, it locates the loop induction variable associated with the IMA load. It identifies its *preheader* and *latch* expressions, which correspond to the initial and bound values of the induction variable, respectively. Second, Magellan unrolls the inner loop twice and compares the preheader from the first iteration with the latch from the second iteration. The nested loop pattern is classified as *irregular* if they do not match. Otherwise, it analyzes the iteration condition of the outer loop: if the outer loop's iteration condition is incremental, the nested loop pattern is identified as *stream-in*; if it is decremental, it is classified as *stream-out*.

With the nested loop pattern identified, the next step is to insert the appropriate prefetch instructions.

3.3 Prefetching Strategy

The core idea behind our prefetch strategy is to aggressively prefetch potential nodes for both current and future loops. This approach is based on the observation that, in sparse applications, inner loops are interrelated—a concept captured by nested loop patterns—rather than isolated. We implement three distinct prefetch strategies to efficiently handle IMAs with different nested loop patterns, as shown

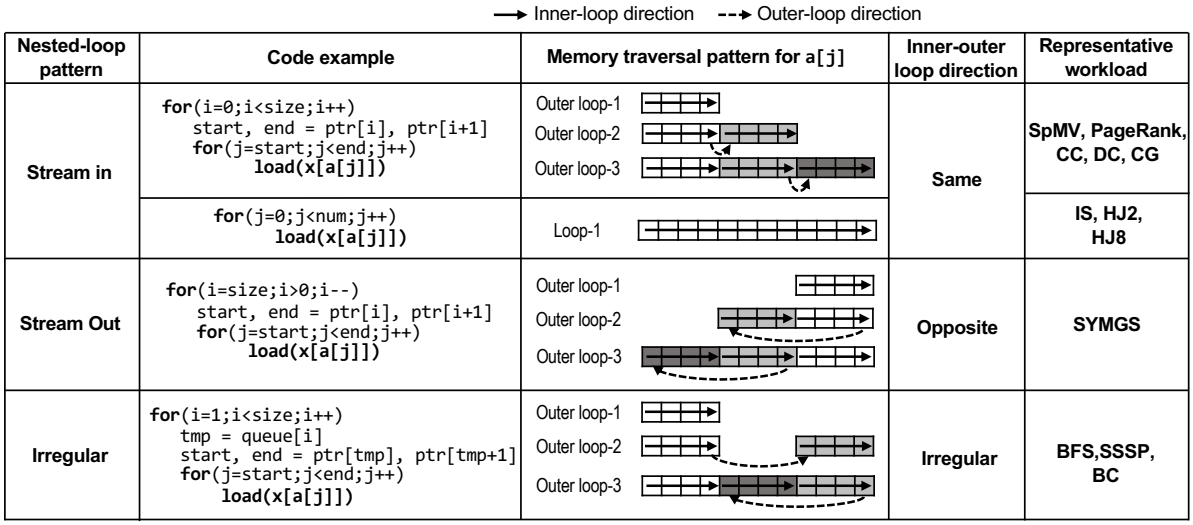

Figure 8: Three nested-loop patterns for sparse applications: stream-in, stream-out, and irregular. Different colors represent different outer loops.

Prefetch strategy	Inner-bound	Inner-free	Opposite inner-free	Outer

Code example	for (i=0;i<size;i++)
	start, end = ptr[i], ptr[i+1]
	for (j=start;j<end;j++)
	load(x[a[j]])
	pref_j = min(j+32, end)
	pref_idx = a[pref_j]
	prefetch(x[pref_idx])
		for (i=0;i<size;i++)
		start, end = ptr[i], ptr[i+1]
		for (j=start;j<end;j++)
		load(x[a[j]])
		pref_j = j+32
		pref_idx = a[pref_j]
		prefetch(x[pref_idx])
			for (i=size;i>0;i--)
			start, end = ptr[i], ptr[i+1]
			for (j=start;j<end;j++)
			load(x[a[j]])
			if j+32 < end
			pref_j = j+32
			else
			pref_j = min(0, start-(j+32-end))
			pref_idx = a[pref_j]
			prefetch(x[pref_idx])
				for (i=1;i<size;i++)
				tmp = queue[i]
				start, end = ptr[tmp], ptr[tmp+1]
				pref_offset = select_node(i+32)
				pref_idx = a[pref_offset]
				prefetch(x[pref_idx])
				for (j=start;j<end;j++)
				load(x[a[j]])

Prefetch position	Inner loop	Inner loop	Inner loop	outer loop
Prefetch scope	Current loop	Current loop + future loop	Current loop + future loop	Future loop
Instruction overhead	High	Medium	High	Low

Figure 9: Four prefetch strategies for indirect memory access. Prefetch-related instructions are noted in red color. Different strategies target different prefetch scopes with different overheads, making them suitable for IMA with specific loop patterns.

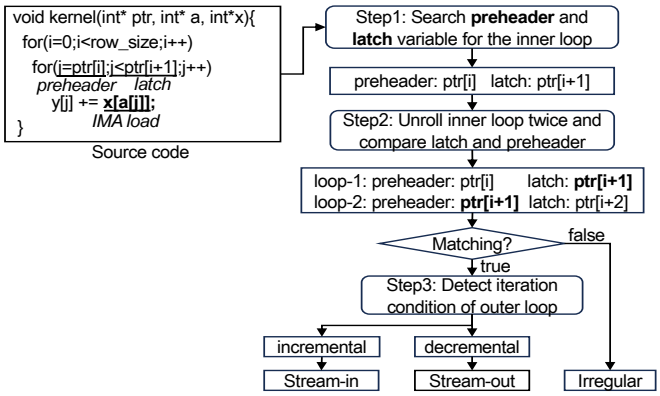

Figure 10: Magellan takes three steps to detect three different nested loop patterns: stream-in, stream-out, and irregular.

in Fig. 9. Each strategy balances the exploitation of prefetch candidates with additional instruction overhead, making it suitable for specific nested loop scenarios.

For IMAs with the stream-in or stream-out nested loop patterns (see Fig.8), the inner-loop induction variable changes regularly between outer loop iterations. This regularity allows us to use the direction of the outer loop to predict node addresses in future loops accurately. Accordingly, we select inner-free prefetching strategies for these patterns, generating prefetches for both current and future loops but in different ways. Since the inner loop shares the same direction as the outer loop in the stream-in pattern, the out-of-loop prefetch index $j + pref\_d$ yields useful requests for future loops. Conversely, in the stream-out pattern, the inner loop moves in the opposite direction to the outer loop. To make out-of-loop prefetches effective when $j + pref\_d$ exceeds the loop bound, we prefetch nodes in the reverse direction, resulting in the opposite inner-free strategy shown in Fig.9.

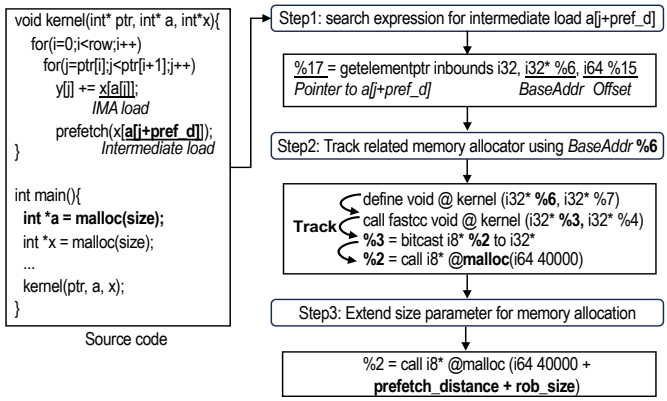

Figure 11: Magellan identifies the memory allocation function associated with prefetched data structures and extends their storage space to avoid memory access violations.

For irregular loop patterns, there are two alternative prefetch strategies. On the one hand, inner-bound prefetching is commonly used to generate conservative prefetch for the current loop. However, it only works well for code with large loop iteration counts when most prefetch indices will not exceed the loop boundary. On the other hand, outer prefetching inserts prefetch instructions in the outer loops, which attempt to prefetch for the future inner loop in advance. Its advantage is that prefetch can now use the whole work from the inner iteration to overlap prefetch latency, resulting in timely prefetch. In sparse applications, where loop iteration counts are small, we opt for outer prefetching for IMAs with irregular loop patterns. We provide a quantitative analysis of this strategy selection in Section 5.12.

3.4 Avoiding Program Faults

Magellan modifies the original program's code by explicitly inserting prefetch-related instructions. Specifically, to perform an IMA prefetch like *x[a[j + pref_d]]*, Magellan inserts an intermediate load for the index *a[j + pref_d]*. Although the prefetch instruction itself is only a hint and will not cause faults on commercial CPUs [30, 47], this intermediate load is a demand request. It can introduce faults if it attempts to access memory out of bounds with an invalid index.

3.4.1 Naïve approach and limitations. A straightforward method to prevent memory access violations is Software Fault Isolation (SFI) [72]. SFI adds explicit boundary checks for each intermediate load, preventing safety violations. However, SFI significantly impacts performance due to the additional instruction overhead. In our evaluation of the SpMV application from the GAP benchmark suite [13], SFI increased the dynamic instruction count by 31% on average and degraded performance by 28%.

3.4.2 Observation: Memory violation occurs at the boundary. We observe that the intermediate load targets sequential access arrays, and memory violations occur only at the array boundaries. Therefore, we must ensure that the farthest access generated by this load instruction during runtime returns a valid value.

3.4.3 Key mechanism: Estimating farthest reachable elements and extending allocation size. Magellan uses two steps to avoid memory violations: (1) analyze the farthest reachable element and (2) extend the memory allocation size of the array to hold this farthest access. There are two factors in this analysis. Consider the IMA code in Fig.11. First, when the last loop iteration is running, and j accesses the last element of array *a*, now the prefetch load *a[j+pref_d]* will exceed the array space. Second, branch misprediction deteriorates this situation. Though speculative accesses on mispredicted paths are not architecturally viable, recently advanced side-channel attacks [33, 44] demonstrate that load instructions on mispredicted paths can be utilized to load illegal data into cache state and transferred by cache-based convert-channels [48, 83]. Therefore, Magellan needs to ensure that loads on the mispredicted path also target the safe memory space. Because if the branch predictor mistakenly predicts the direction of the inner loop branch, j would increase mistakenly, and prefetched index *j+pref_d* would increase accordingly.

Second, our compiler pass tracks the allocation function associated with array *a* and extends its size accordingly, as shown in Fig. 11. LLVM typically transforms the intermediate load *a[j + pref_d]* into a *GetElementPtr* (GEP) instruction in the form *GEP {BaseAddress, Offset}*. The *BaseAddress* variable refers to the array's base address, and the *Offset* variable indexes into the array. We use *BaseAddress* as a starting point and search backward through its preceding instructions by examining the source operands until we locate the memory allocation function (e.g., *malloc* or *calloc*). We then increase the size parameter by *prefetch_distance + rob_size* to ensure that each intermediate load *a[j + pref_d]* returns valid data. Since the data footprints of sparse applications are typically very large, the additional memory space consumed by Magellan is negligible – averaging only 0.0036% of the total memory cost – as discussed in Section 5.6.

To avoid out-of-bound memory accesses, our optimization is applied only when the memory allocation site associated with the prefetched load can be precisely identified and safely extended. Some programs may involve prefetched loads originating from multiple pointers allocated via separate *malloc()* calls. In such cases, if any allocation site cannot be accurately tracked, such as when prefetched data structures are allocated externally, our optimization is not applied to preserve program correctness.

Specifically, we address external allocations by examining the linkage type of functions to determine if they are externally callable and by checking the attributes of prefetched data structures to see if they are external variables. Another challenging scenario occurs when pointers propagate through memory. Here, our LLVM pass tracks memory dependencies using *AliasSetTracker* [74] and *DominatorTreeAnalysis* [17] from the LLVM framework. These analyses help perform pointer aliasing and identify the last store operation preceding the prefetched load. However, since our approach relies solely on static analysis, it may encounter difficulties in complex aliasing situations, such as pointer accesses that are conditional on runtime branches. Additionally, our optimization deliberately excludes calls to *memcpy()* and *memmove()* when the prefetched load targets a derived memory region, since these functions strictly copy valid data and do not require extended sizes for prefetch safety.

Table 1: Evaluation platform configuration.

Architecture	AArch64	x86	x86
Platform	GEM5	Intel i5-7500	Intel E5-2660
Processor	ARMv8	Kabylake	Sandy Bridge
Frequency	2.5GHz	3.4GHz	2.2GHz
L1 D-Cache	32KB	32KB	512KB
L2 Cache	1MB	256KB	4MB
L3 Cache	/	6MB	40MB
Memory	16GB	16GB	192GB

Table 2: The kernels used in this study.

Benchmark	Source	IMA type		Nested Loop Pattern		
		Local	Global	Stream-in	Stream-out	Irregular
SpMV	GAP [13]	✓	✓	✓		
PR	GAP [13]	✓	✓	✓		
BFS	GAP [13]	✓	✓	✓		✓
SSSP	GAP [13]	✓	✓	✓		✓
BC	GAP [13]	✓	✓	✓		✓
CC	GAP [13]	✓	✓	✓		
TC	GAP [13]	✓	✓	✓		✓
DC	GraphBIG [58]	✓	✓	✓		
SYMGS	HPCG [25]	✓	✓	✓	✓	
IS	NAS [7]	✓		✓		
CG	NAS [7]	✓	✓	✓		
HJ2	HashJoin [11]	✓		✓		
HJ8	HashJoin [11]	✓		✓		
RandAcc	HPCC [51]	✓		✓		

Table 3: Real-world graph datasets for evaluation

Matrix	Vertices	Edges	Domain
road_usa (RU)	23,947,347	57,708,624	Road Graph
com-liveJournal (CL)	3,997,962	69,362,378	Social Network
soc-pokec (SP)	1,632,803	30,622,564	Social Network
asia_osm (AO)	11,950,757	25,423,206	Street Network

4 Experimental Methodology

4.1 System Configurations

We implement the Magellan prefetcher as an LLVM IR pass [45], compatible with the Clang compiler. Table 1 summarizes the specifications of the platforms used in our evaluation. We assess Magellan's effectiveness on two commercial hardware platforms: a client platform featuring a 3.4 GHz Intel Core i5-7500 processor, and a server platform equipped with a 2.2 GHz Intel Xeon E5-2660 processor. Both platforms support out-of-order execution along with software and hardware prefetching mechanisms.

We benchmark Magellan against three advanced software prefetching methods: SW Prefetch [4], APT-GET [38], and Intel OneAPI [65] (version 2024.1). APT-GET, one of the latest profile-guided software prefetchers, requires profiling input data for several minutes and recompiles the benchmarks using the collected profiling information to optimize prefetch parameters. The Intel OneAPI compiler improves application performance via automatic software prefetch optimization enabled by the *-qopt-prefetch* option.

Additionally, we use the gem5 cycle-accurate simulator [15] to compare Magellan against four state-of-the-art hardware prefetchers: IPCP [66], Berti [61], IMP [84], and DMP [29], as well as one software-hardware co-designed prefetcher, Event-trigger [3]. The simulated system uses Intel Skylake parameters [26]. Performance results exclude initialization costs, such as reading input graphs and setting up data structures. For evaluations on actual hardware, algorithms execute for at least one minute to minimize initialization overhead. In gem5 simulations, we use the region-of-interest (ROI) utility to isolate and profile only the core algorithmic execution.

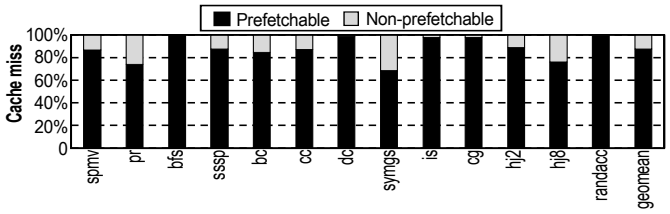

Figure 12: Classification of cache miss into two types: prefetchable (IMA miss) and non-prefetchable (other miss).

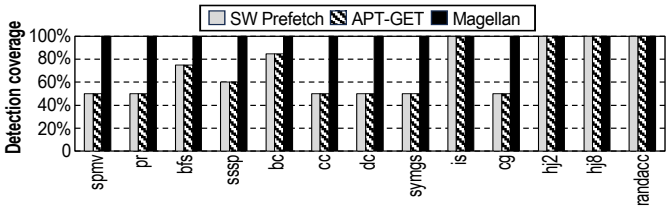

Figure 13: Indirect pattern detection coverage for SW Prefetch [4], APT-GET [38] and Magellan.

4.2 Benchmarks

We evaluate Magellan using 14 diverse benchmark applications spanning graph analytics, sparse linear algebra, and database domains. This broad selection underscores the general applicability of indirect memory access (IMA) patterns. Specifically, we use graph algorithms such as Sparse Matrix-Vector Multiplication (SPMV), PageRank (PR), Breadth-First Search (BFS), Single-Source Shortest Path (SSSP), Betweenness Centrality (BC), Triangle Counting (TC), Connected Components (CC), and Degree Centrality (DC), drawn from the GAP [13] and GraphBIG [58] benchmarks. Sparse linear algebra applications include the Symmetric Gauss-Seidel Smoother (SYMGS) from HPCG [25] and RandomAccess (RANDACC) from HPCC [51]. Integer Sort (IS) and Conjugate Gradient (CG), relevant to computational fluid dynamics, are selected from the NAS parallel benchmarks [7]. Database system behavior is represented by Hashjoin 2EPB (HJ2) and Hashjoin 8EPB (HJ8) from prior studies [11]. Table 2 summarizes the indirect access and nested loop patterns across these workloads. Real-world datasets from graph collections [24], detailed in Table 3, are used for graph algorithms. For IS and CG, we evaluate using the CLASS-B scale (33MB).

5 Evaluation

5.1 Prefetch Potential

We first evaluate the upper bound for the performance potential by software prefetching to understand the capability of the ideal Magellan prefetcher. Ideally, prefetch requests should cover all cache misses resulting from indirect memory accesses, and the associated DRAM accesses would then become cache hits. We use Cachegrind [62] to evaluate the percentage of cache misses triggered by IMA load instructions on the non-prefetching baseline. Fig. 12 shows that, on average, 87.6% of total cache misses are IMA misses, which can be seen as potential candidates for the Magellan prefetcher.

```
for (auto q_iter = queue.begin(); q_iter < queue.end(); q_iter++) {
      NodeID u = *q_iter;
      for (NodeID v : g.out_neigh(u)) {
        NodeID curr_val = parent[v];
        ...
      }
    }
```

Figure 14: Global IMA of BFS code from GAP benchmark[13].

5.2 IMA Detection Coverage

First, effective prefetching requires accurately identifying the underlying indirect memory access (IMA) patterns within programs. Fig.13 compares the detection coverage of different prefetching techniques. All techniques demonstrate accurate detection for straightforward kernels such as IS, HJ2, and HJ8, where the IMA pattern appears in a simple, single-level loop structured as *x[a[i]]*. However, for more complex graph applications like BFS and SSSP, SW Prefetch and APT-GET exhibit limited detection coverage, which reduces their effectiveness in prefetching. As shown in Figure 14, these graph applications frequently involve nested two-level loops: the outer loop selects a vertex, and the inner loop iterates over all edges connected to that vertex. SW Prefetch and APT-GET terminate their backward search for loads such as *parent[v]* when encountering the loop variable *v*, treating *parent[v]* as a stream load. However, the loop variable *v* depends on the outer-loop variable *u* through the loop's iteration conditions. Magellan addresses this issue by explicitly analyzing iteration conditions and constructing dependency graphs across nested loops. This enables Magellan to capture cross-loop dependencies, identifying the load of *parent[v]* as a global IMA and appropriately inserting indirect prefetch instructions.

5.3 Overall Performance

Fig.15 summarizes Magellan's performance across all workloads on the Kabylake and Sandy Bridge platforms. Performance results are presented as normalized speedups relative to the original code without software prefetching. On the Kabylake platform, Magellan achieves an average speedup (geomean) of 1.2×, with a maximum of 2.5× compared to the baseline. On the Sandy Bridge platform, Magellan achieves an average speedup of 1.1×, with a maximum of 1.6× observed in the HJ8 application. Magellan generally provides greater benefits on the client platform (Kabylake) compared to the server platform (Sandy Bridge). This difference likely arises from the server's more aggressive memory hierarchy, characterized by higher associativity and larger cache sizes, which reduces the number of cache misses.

Applications such as **SpMV, PR, CC, and DC** exhibit significant speedups with Magellan, achieving a geomean speedup of 1.2× and up to 1.4× compared to the baseline. These applications feature *stream-in* nested-loop structures, enabling Magellan to identify additional prefetch opportunities by employing out-of-loop prefetching strategies. Figure 16 illustrates that Magellan reduces cache misses by an average of 89% compared to the baseline, whereas SW Prefetch achieves only a 36% reduction due to its inner-bound prefetch strategy. Additionally, Figure 17 shows that Magellan lowers normalized instruction counts from 45% to 29% compared to SW Prefetch, primarily by eliminating boundary checks.

Applications such as **BFS, SSSP, and BC** display similar performance trends, with Magellan achieving an average speedup of 1.2× over the baseline. However, these applications involve more complex indirect memory accesses (IMAs) and nested-loop patterns. For example, in BFS, traversals of *Offset* demonstrate local IMAs with a stream-in loop pattern, whereas traversals of *Edgelist* exhibit global IMAs with irregular loop patterns due to dynamic vertex processing order. Magellan strategically generates out-of-loop prefetches for *Offset* and uses an outer-loop prefetching for data like *Visited*.

However, performance degradation is observed in some **BC** scenarios due to two primary reasons. First, BC involves significantly more complex IMAs, featuring 13 distinct indirection loads, which can lead to frequent prefetch insertions interfering with regular memory requests. Second, static compiler-time prefetch scheduling struggles with complex access patterns, highlighting the potential benefits of a dynamic feedback-based system to fine-tune Magellan's prefetch strategy.

In contrast, **TC** achieves a modest speedup of 1.06× with Magellan due to good data locality and frequent data reuse, making it more compute-bound than memory-bound. Analysis using VTune Profiler reveals that only 15% of pipeline slots are impacted by memory latency, limiting prefetch benefits. The **SYMGS** application differs from others, featuring two runtime phases: forward incremental traversal followed by backward traversal of matrix rows. Magellan applies distinct inner-free prefetch strategies in each phase, effectively exploiting prefetch opportunities and achieving significant speedups (1.2×). For simpler applications such as **IS, CG, HJ2, and HJ8**, Magellan and SW Prefetch both achieve notable performance improvements due to IMAs typically appearing in straightforward, single-level loops (e.g., *x[a[i]]*) with sufficient iteration counts for timely prefetch generation.

Finally, APT-GET [38] achieves an average speedup of 1.13× over SW Prefetch [4] due to its profile-guided prefetch optimization. Nonetheless, Magellan still outperforms APT-GET by an average of 1.14× by leveraging deeper loop semantics. Incorporating APT-GET's profile-based tuning approach could further enhance Magellan's performance by optimizing prefetch parameters across diverse applications.

5.4 Magellan versus Hardware Prefetchers

Beyond software prefetchers, we also compare Magellan with five state-of-the-art hardware prefetchers (i.e., IPCP [66], Berti [61], IMP [84], and DMP [29]) and one software-hardware co-designed prefetcher named Event-trigger [3]. The key benefit of Magellan over hardware prefetchers is its ability to accommodate different program behaviors without additional hardware costs. Fig.18 shows that Magellan outperforms the non-prefetching baseline by 1.7× on average, compared with 1.8× speedup by the best of hardware prefetchers. The results show two observations. First, for most applications, compared with the state-of-the-art hardware prefetchers, Magellan achieves competitive performance improvements without requiring any hardware changes to current processors. Second HJ2, HJ8, RANDACC applications have more complicated IMA patterns due to the hash function with the form of *x[hash(a[i])]*. For this type of IMA, it is difficult for hardware prefetchers to accurately detect the IMA pattern, considering the various possible implementations

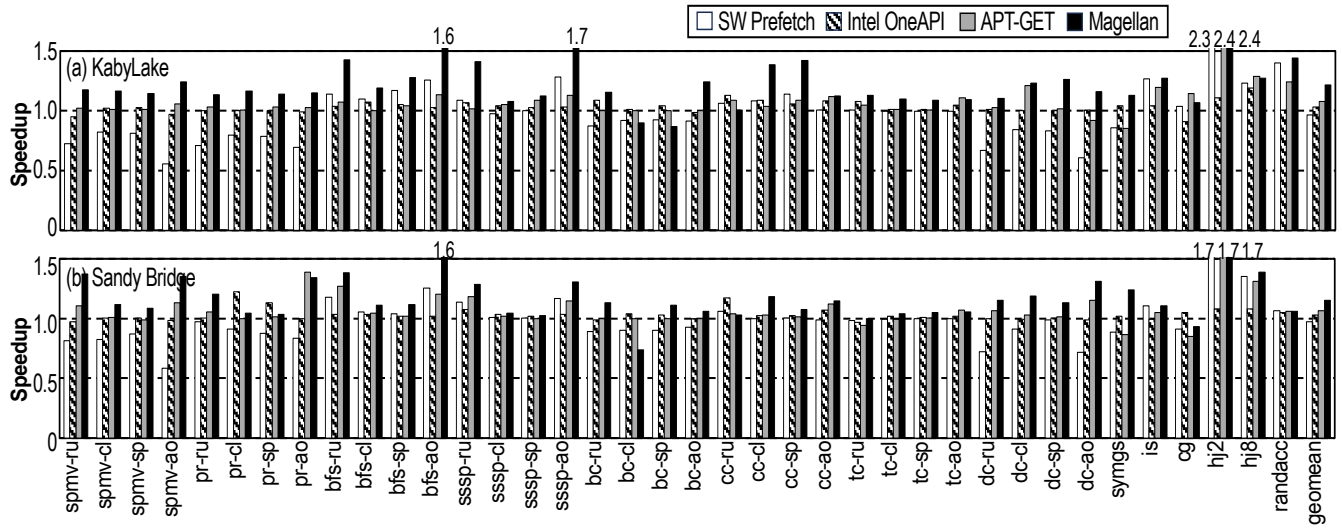

Figure 15: SW Prefetch [4], Intel OneAPI [65], APT-GET [38] and Magellan performance on Kabylake (top) and Sandy Bridge (bottom). Performance is normalized to the baseline with no software prefetcher.

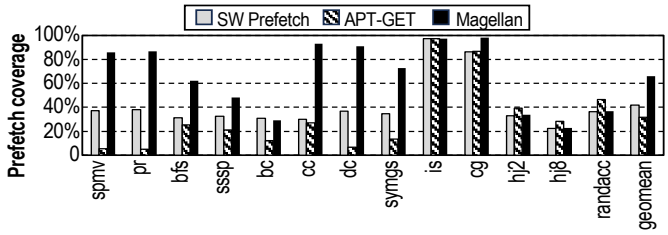

Figure 16: Prefetch coverage by SW Prefetch, APT-GET and Magellan.

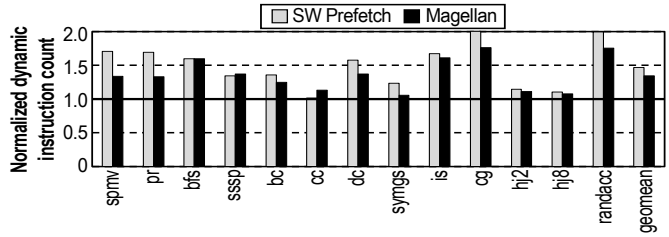

Figure 17: Dynamic instruction count of SW Prefetch and Magellan. Results normalized to non-prefetching baseline.

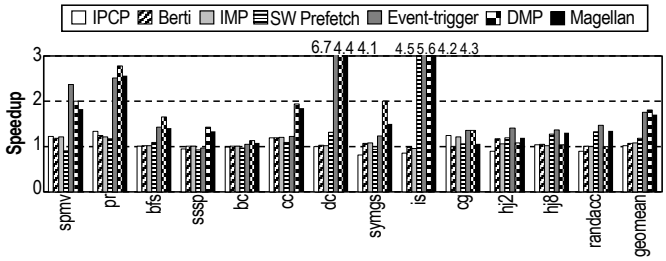

Figure 18: Performance of Magellan, SW Prefetch and five state-of-the-art hardware prefetchers over non-prefetching baseline.

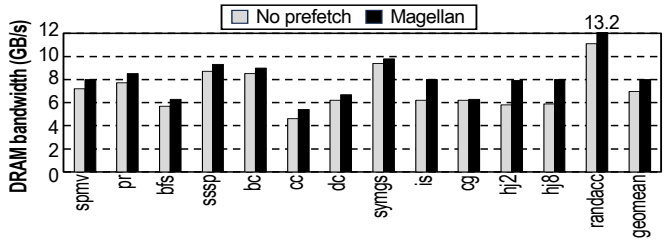

Figure 19: Memory bandwidth by no-prefetch and Magellan.

of the hash function, which results in a restricted performance speedup. In contrast, Magellan uses knowledge about program behavior and data distribution at compile time to accurately detect this complicated IMA pattern.

5.5 Memory Bandwidth Usage

To further evaluate Magellan's effect on memory optimization, Fig.19 depicts the DRAM bandwidth usage of Magellan, which slightly boosts bandwidth, reaching an average of 1.1× bandwidth than the non-prefetching baseline. This result shows two observations. First, prefetching insertion indeed increases memory bandwidth, and performance improvement is a trade-off that comes at the cost of consuming more bandwidth. An adaptive mechanism [31] that dumps prefetching requests based on bandwidth usage as feedback can improve this situation. Second, Magellan does not impose significant extra pressure on the memory bandwidth, which means that most prefetching requests turn into useful memory accesses for demand loads. Therefore, with highly accurate detection and generation of most indirect patterns, Magellan greatly improves memory access efficiency, which explains its significant speedup from the computer system's perspective.

5.6 Memory Storage Cost

Magellan ensures that the addresses of intermediate loads always return valid values by explicitly expanding the size of the relevant

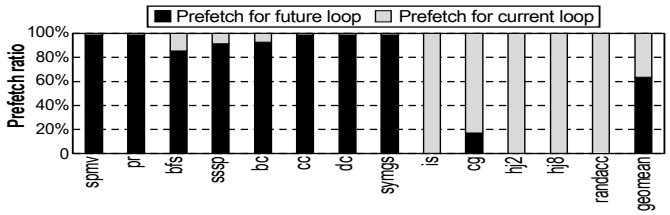

Figure 20: Classification of prefetch requests issued by Magellan into targeting future loop and target current loop.

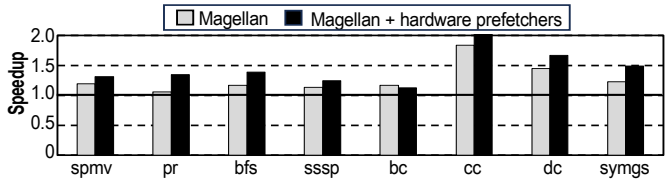

Figure 21: Performance of Magellan with hardware prefetchers disabled and enabled.

data structures. The size of the extended space is proportional to three factors: the number of IMAs (max number in our evaluation is 13 for BC application), the ROB size (224/168 for Kabylake / SandyBridge), and the prefetch look-ahead distance (set as 32 in Magellan configuration). The cost of extended memory storage is negligible because sparse applications typically have large data sets, much larger than the LLC size. We estimate the average storage cost per application is 1486 Bytes (0.0036% of total memory on average).

5.7 Prefetch Target Analysis

To better understand how Magellan exploits the prefetch opportunity, Fig. 20 classifies the prefetch requests issued by Magellan into two types: targeting the current loop and targeting the future loop. Overall, this figure reveals that Magellan generates prefetches from both the current loop and the future loop, but its behavior differs across different applications. For sparse and graph applications, due to the sparse distribution of data input, most loops have a small iteration count. In this case, Magellan focuses on prefetching candidates from the future loop to boost performance. In contrast, for applications with a large iteration count, such as IS and CG, Magellan mainly exploits prefetch candidates from the current loop.

5.8 Influence of the Hardware Prefetcher

Fig.21 presents the speedups of Magellan to a non-prefetching baseline with different hardware prefetcher configurations. We can observe that hardware prefetchers further improve Magellan's performance by 1.13× on average. The reason is that using Magellan and hardware prefetchers together can help cover a broader range of access patterns. Specifically, hardware prefetchers cover regular stream accesses, and Magellan covers irregular indirect accesses.

5.9 Outer-Prefetching Degree

Figure 22 illustrates the impact of different outer-prefetching degrees on overall performance, normalized to a baseline without prefetching. Two key observations emerge: first, there can be up to a 40% performance gap between the best and worst prefetching

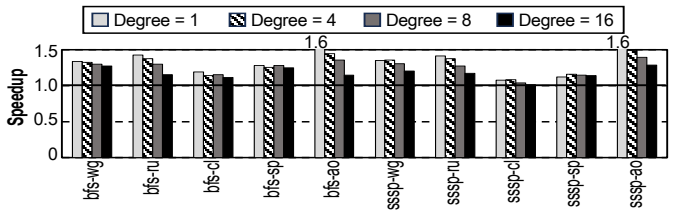

Figure 22: Performance of Magellan with different outer-prefetch degrees. Results are normalized to no-prefetch.

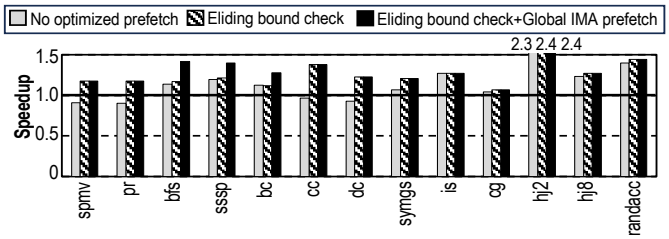

Figure 23: Effect of eliding bound check and global IMA prefetch on Magellan's overall performance.

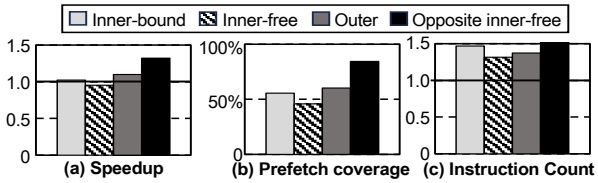

Figure 24: (a) Speedup, (b) prefetching coverage, and (c) dynamic instruction count for different prefetching strategies for IMA with stream-out loop pattern.

configurations. This disparity arises from variations in the number of memory requests issued by different prefetching degrees, resulting in varying levels of instruction overhead. Second, in the SSSP application with the *com-LiveJournal(cl)* dataset, a prefetching degree of 4 yields the highest performance. Nevertheless, since most scenarios attain optimal results with a prefetching degree of 1, we adopt *degree=1* in our design, providing a balanced trade-off between coverage and instruction overhead.

5.10 Effect: Bound Checks and Global IMA Prefetch

To analyze the effects of different optimizations on Magellan's overall performance, we design three comparative schemes: (a) non-optimized prefetching, (b) prefetching with bound-check-removing, (c) prefetching with bound-check-removing and global IMA prefetch. Fig. 23 shows the comparison results. First, prefetching without any optimization only provides limited speedup, and for sparse applications(e.g., SPMV), it even causes a slowdown. These applications consume extra instructions to generate prefetch requests, but most of them are bounded by loop boundaries. Removing bound checks increases the performance by 11% because it removes bound check instructions and exploits more prefetch potentials. Moreover, global IMA prefetch can further increase 15% performance for graph applications by identifying more prefetch candidates.

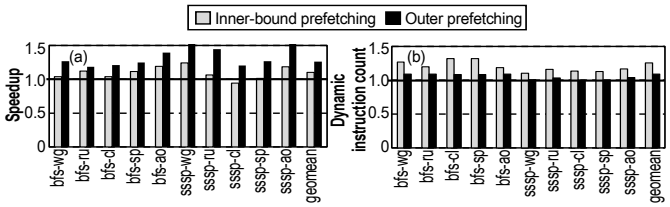

Figure 25: Performance (a) and dynamic instruction count (b) of two different prefetch strategies for IMA with irregular nested loop patterns.

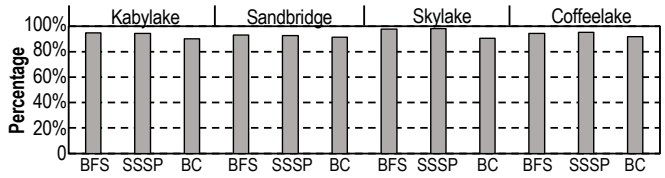

Figure 26: Percentage of matrices in which using outer prefetching helps outperform inner prefetching. We test matrices that have more than 10 million non-zero values from SuiteSparse[24].

5.11 Strategy Selection for Stream-out Loop

The stream-out loop pattern is unique, featuring opposite directions for the inner loop and outer loop. We measure the performance of IMA in stream-out loops for SYMGS application with different prefetching strategies. As shown in Fig.24(a), the opposite inner-free strategy achieves the highest speedup of 1.3× normalized to no-prefetch baseline. This is because the opposite inner-free strategy distinguishably generates prefetches targeting the current and future loops, resulting in the best prefetch coverage in Fig.24(b). Besides, it only costs a reasonable extra instruction overhead compared with other strategies in Fig.24(c). Thus, our design uses an opposite inner-free strategy to prefetch IMAs in stream-out loops.

5.12 Prefetch for Irregular Nested Loop Patterns

Two alternative strategies for prefetching IMA with an irregular nested loop pattern are inner-bound prefetching and outer prefetching, which focus on the prefetch opportunity for each current loop and future loop individually. Fig. 25(a) compares the performance of the inner-bound and outer prefetching strategies. In general, outer prefetching outperforms inner-bound prefetching by 1.2× on average. In addition, we also compare the instruction overhead of these two strategies relative to the non-prefetching baseline, as shown in Fig. 25 (b). Because inner-bound prefetching performs prefetch for every inner-loop iteration, it incurs 15% more instructions than outer prefetching. We further test outer and inner prefetching on applications with irregular loop patterns (BFS,SSSP,BC) using all matrices that have more than 10 million non-zeros from the SuiteSparse collection [24] (a total of 209) on four different machines. As shown in Fig. 26, the outer prefetching strategy yields better speedups on most matrices across different machines. Thus, we use the outer prefetching strategy for irregular IMA patterns.

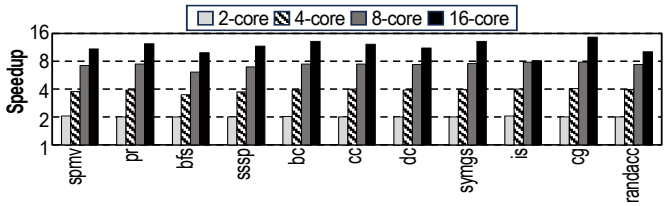

Figure 27: Performance of Magellan for 2 to 16-core system. Results are normalized to a 1-core system with Magellan.

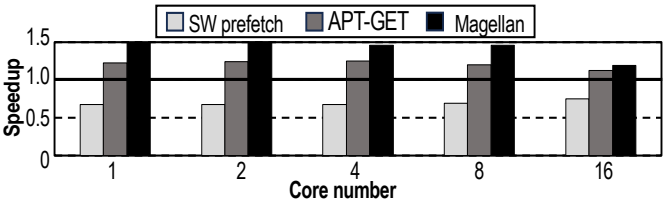

Figure 28: Multi-core Speedups of SW Prefetch, APT-GET, and Magellan on SPMV over non-prefetching baseline.

5.13 A Discussion on Scalability

To see whether Magellan continues to provide performance benefits as the number of cores increases, we experimented with Magellan, ranging from 2 to 16 cores. Performance is normalized to a 1-core system with Magellan prefetcher. As shown in Fig. 27, the performance of Magellan improves as the number of cores increases. For example, the performance of SPMV for eight cores achieves nearly 7.2× speedup compared to the performance of 1 core. In addition, we observe that increasing the number of cores to around 16 has a significant performance drop compared to the 1-core system. This performance drop can be mainly attributed to the increasing contention in DRAM memory bandwidth, which introduces longer DRAM access latencies. In that case, Magellan should adjust its prefetch aggressiveness by reducing prefetch levels and degrees, which is the future direction of this study.

Fig.28 shows speedups of different software prefetchers from 1 to 16 cores on *spmv* application. Overall, Magellan outperforms competing prefetchers on systems with different core numbers. Besides, in the 16-core system, the performance gap between APT-GET and Magellan becomes smaller because the per-core available DRAM bandwidth decreases, limiting the prefetch benefits.

6 Related Work

Indirect memory accesses are widely deployed in several application domains, including graph analytics [2], machine learning [20], High-Performance Computing [50], and Warehouse-Scale (WSC) applications [39]. Many important classes of algorithms, including graph algorithms (e.g., GAP benchmark [13]), machine learning(e.g., sparse CNN [34] , graph neural network [42]), HPC (e.g., global dot products, sparse triangular solve [25]), WSC [39] (e.g. web search, knowledge graph) and general-purpose applications (e.g., *mcf* in SPEC CPU2006 [32]) share this kind of memory access pattern. Despite their prevalence, current commercial CPUs offer suboptimal performance that can be further improved [63, 81]. This section discusses works that alleviate this memory bottleneck.

Software prefetching uses the compiler to analyze the source code and insert prefetch instructions statically [5]. Previous work in software prefetching targets simple access patterns such as streaming or stride access [54]. More recently, several works have targeted more complex access patterns, such as pointer chasing, hash join, and link data structures [19, 40, 49, 71], but not the indirect memory access patterns discussed in this paper. Recently, three new compiler techniques for indirection patterns have been proposed: SW Prefetch [4], APT-GET [38], and RPG^2 [85]. Because they do not efficiently leverage control flow behavior, they have limited applicability for sparse applications with combinations of different indirection patterns. Compared to SW Prefetch, APT-GET, and RPG^2, these use runtime information to optimize prefetcher parameters, such as prefetch distance, differently. APT-GET uses the Perf tool to perform profiling and recompile the program accordingly. In contrast, RPG^2 uses the BOLT [67] tool to dynamically rewrite the binary file and tune the prefetch parameters. These mechanisms can be applied to our design to boost performance further.

Hardware prefetching uses dedicated hardware to detect preassumed memory access patterns and issue prefetching requests in advance to hide the memory latency. The hardware implicitly manages hardware prefetching, and for this reason, it is also called automatic prefetching. Hardware prefetchers are more challenging to implement than software approaches because they detect target access patterns at runtime without software assistance. The prediction of hardware prefetchers usually attempts to capture regular spatial [6, 9, 14] or temporal [8, 37, 41, 52, 78, 79] history patterns, which are not applicable for workloads with irregular indirection accesses. Specialized IMA hardware prefetchers such as IMP [84] and DMP [29] require additional custom logic to detect and prefetch complex indirection patterns.

Runahead [57, 59, 60] utilizes the CPU's memory stall time to prefetch useful data. Recently, Vector Runahead [59] has been proposed to target indirect memory access, which speculatively reorders scalar operations in vector format to prefetch multiple indirect accesses in parallel. However, unlike software prefetching techniques, runahead execution requires heavy modification of the CPU core design, which limits its applicability in commercial CPUs.

Graph accelerators [1, 12, 55, 56, 69, 70] have also been proposed, using pipelining data [55, 69], intelligent caching [12, 56], and near/in-memory processing [1, 70], to improve memory access. These architectures can use our prefetcher to enhance performance.

7 Conclusions

We introduced Magellan, a loop-guided software prefetcher for indirect memory accesses (IMAs) in irregular applications. Magellan leverages loop behavior to efficiently detect and generate prefetches for complex IMA patterns. It increases prefetch opportunities in current and future loop iterations by selecting different prefetching strategies based on loop structures. Additionally, Magellan uses graphs that incorporate induction variables and dependent loads from multiple levels of nested loops to uncover more complex memory access patterns. These innovations enable Magellan to achieve significant performance speedups across various applications and datasets. Thus, Magellan offers a high-performance and practical solution to alleviate memory bottlenecks.

Acknowledgments

We thank the shepherd and anonymous reviewers for their valuable comments and feedback. This research was supported in part by the National Key Research and Development Program of China (No. 2022YFB4500500), the National Natural Science Foundation of China (No. 62302381 and 62088102), and funding from the Laboratory for Advanced Computing and Intelligence Engineering. The authors from Xi'an Jiaotong University are affiliated with the State Key Laboratory of Human-Machine Hybrid Augmented Intelligence, the National Engineering Research Center for Visual Information and Applications, and the Institute of Artificial Intelligence and Robotics. Tian Xia is also affiliated with Laboratory for Advanced Computing and Intelligence Engineering, Wuxi, China. Additionally, Prashant J. Nair and Mieszko Lis acknowledge support from Canada's Natural Sciences and Engineering Research Council Discovery Grants RGPIN-2019-05059 and RGPIN-2023-05380, respectively.

References

[1] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. A scalable processing-in-memory accelerator for parallel graph processing. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. 105–117.

[2] Sam Ainsworth and Timothy M Jones. 2016. Graph prefetching using data structure knowledge. In *Proceedings of the 2016 International Conference on Supercomputing*. 1–11.

[3] Sam Ainsworth and Timothy M Jones. 2018. An event-triggered programmable prefetcher for irregular workloads. *ACM Sigplan Notices* 53, 2 (2018), 578–592.

[4] Sam Ainsworth and Timothy M Jones. 2019. Software prefetching for indirect memory accesses: A microarchitectural perspective. *ACM Transactions on Computer Systems (TOCS)* 36, 3 (2019), 1–34.

[5] 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*. 513–526.

[6] Jean-Loup Baer and Tien-Fu Chen. 1991. An effective on-chip preloading scheme to reduce data access penalty. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*. 176–186.

[7] David H Bailey, Eric Barszcz, Leonardo Dagum, and Horst D Simon. 1993. NAS parallel benchmark results. *IEEE Parallel & Distributed Technology: Systems & Applications* 1, 1 (1993), 43–51.

[8] Mohammad Bakhshalipour, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. 2018. Domino temporal data prefetcher. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 131–142.

[9] Mohammad Bakhshalipour, Mehran Shakerinava, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. 2019. Bingo spatial data prefetcher. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 399–411.

[10] Vignesh Balaji, Neal Crago, Aamer Jaleel, and Brandon Lucia. 2021. P-opt: Practical optimal cache replacement for graph analytics. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 668–681.

[11] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M Tamer Özsu. 2013. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 362–373.

[12] Abanti Basak, Shuangchen Li, Xing Hu, Sang Min Oh, Xinfeng Xie, Li Zhao, Xiaowei Jiang, and Yuan Xie. 2019. Analysis and optimization of the memory hierarchy for graph processing workloads. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 373–386.

[13] Scott Beamer, Krste Asanović, and David Patterson. 2015. The GAP benchmark suite. *arXiv preprint arXiv:1508.03619* (2015).

[14] Rahul Bera, Anant V Nori, Onur Mutlu, and Sreenivas Subramoney. 2019. Dspatch: Dual spatial pattern prefetcher. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 531–544.

[15] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. 2011. The gem5 simulator. *ACM SIGARCH computer architecture news* 39, 2 (2011), 1–7.

[16] Paolo Boldi and Sebastiano Vigna. 2004. The webgraph framework I: compression techniques. In *Proceedings of the 13th international conference on World Wide Web*.

595–602.
[17] Adam L. Buchsbaum, Loukas Georgiadis, Haim Kaplan, Anne Rogers, Robert E. Tarjan, and Jeffery R. Westbrook. 2008. Linear-Time Algorithms for Dominators and Other Path-Evaluation Problems. *SIAM J. Comput.* 38, 4 (Nov. 2008), 1533–1573. doi:10.1137/070693217
[18] Aydin Buluc and John R Gilbert. 2008. On the representation and multiplication of hypersparse matrices. In *2008 IEEE International Symposium on Parallel and Distributed Processing*. IEEE, 1–11.
[19] Shimin Chen, Anastassia Ailamaki, Phillip B Gibbons, and Todd C Mowry. 2007. Improving hash join performance through prefetching. *ACM Transactions on Database Systems (TODS)* 32, 3 (2007), 17–es.
[20] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 578–594.
[21] Tien-Fu Chen and Jean-Loup Baer. 1995. Effective hardware-based data prefetching for high-performance processors. *IEEE transactions on computers* 44, 5 (1995), 609–623.
[22] Jamison Collins, Suleyman Sair, Brad Calder, and Dean M Tullsen. 2002. Pointer cache assisted prefetching. In *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*. 62–73.
[23] JD Collins, Hong Wang, DM Tullsen, C Hughes, Yong-Fong Lee, D Lavery, and JP Shen. 2001. Speculative precomputation: long-range prefetching of delinquent loads. In *Proceedings of 28th Annual International Symposium on Computer Architecture*. IEEE Computer Society, 14–25.
[24] Timothy A Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)* 38, 1 (2011), 1–25.
[25] Jack Dongarra, Michael A Heroux, and Piotr Luszczek. 2015. HPCG benchmark: a new metric for ranking high performance computing systems. *Knoxville, Tennessee* 42 (2015).
[26] Jack Doweck, Wen-Fu Kao, Allen Kuan-yu Lu, Julius Mandelblat, Anirudha Rahatekar, Lihu Rappoport, Efraim Rotem, Ahmad Yasin, and Adi Yoaz. 2017. Inside 6th-generation intel core: New microarchitecture code-named skylake. *IEEE Micro* 37, 2 (2017), 52–62.
[27] Babak Falsafi and Thomas F Wenisch. 2022. *A primer on hardware prefetching*. Springer Nature.
[28] Yu Feng, Gunnar Hammonds, Yiming Gan, and Yuhao Zhu. 2022. Crescent: taming memory irregularities for accelerating deep point cloud analytics. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. 962–977.
[29] Gelin Fu, Tian Xia, Zhongpei Luo, Ruiyang Chen, Wenzhe Zhao, and Pengju Ren. 2024. Differential-Matching Prefetcher for Indirect Memory Access. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 439–453.
[30] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. 2016. Prefetch side-channel attacks: Bypassing SMAP and kernel ASLR. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. 368–379.
[31] Wim Heirman, Kristof Du Bois, Yves Vandriessche, Stijn Eyerman, and Ibrahim Hur. 2018. Near-side prefetch throttling: Adaptive prefetching for high-performance many-core processors. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*. 1–11.
[32] John L Henning. 2006. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News* 34, 4 (2006), 1–17.
[33] Mathé Hertogh, Sander Wiebing, and Cristiano Giuffrida. 2024. Leaky Address Masking: Exploiting Unmasked Spectre Gadgets with Noncanonical Address Translation. In *2024 IEEE Symposium on Security and Privacy (SP)*. 3773–3788. doi:10.1109/SP54263.2024.00158
[34] Torsten Hoefler, Dan Alistarh, Tal Ben-Nun, Nikoli Dryden, and Alexandra Peste. 2021. Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks. *Journal of Machine Learning Research* 22, 241 (2021), 1–124.
[35] Tatsushi Inagaki, Tamiya Onodera, Hideaki Komatsu, and Toshio Nakatani. 2003. Stride prefetching by dynamically inspecting objects. *ACM SIGPLAN Notices* 38, 5 (2003), 269–277.
[36] Intel® 64 and IA-32 Architectures Optimization Reference Manual. [n. d.]. https://www.intel.com/content/www/us/en/content-details/671488/intel-64-and-ia-32-architectures-optimization-reference-manual.html.
[37] Akanksha Jain and Calvin Lin. 2013. Linearizing irregular memory accesses for improved correlated prefetching. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. 247–259.
[38] Saba Jamilan, Tanvir Ahmed Khan, Grant Ayers, Baris Kasikci, and Heiner Litz. 2022. Apt-get: Profile-guided timely software prefetching. In *Proceedings of the Seventeenth European Conference on Computer Systems*. 747–764.
[39] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2015. Profiling a warehouse-scale computer. In *Proceedings of the 42nd annual international symposium on computer architecture*. 158–169.
[40] Magnus Karlsson, Fredrik Dahlgren, and Per Stenstrom. 2000. A prefetching technique for irregular accesses to linked data structures. In *Proceedings Sixth International Symposium on High-Performance Computer Architecture. HPCA-6 (Cat. No. PR00550)*. IEEE, 206–217.
[41] Jinchun Kim, Seth H Pugsley, Paul V Gratz, AL Narasimha Reddy, Chris Wilkerson, and Zeshan Chishti. 2016. Path confidence based lookahead prefetching. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1–12.
[42] Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* (2016).
[43] Onur Kocberber, Boris Grot, Javier Picorel, Babak Falsafi, Kevin Lim, and Parthasarathy Ranganathan. 2013. Meet the walkers: Accelerating index traversals for in-memory databases. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. 468–479.
[44] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *2019 IEEE Symposium on Security and Privacy (SP)*. 1–19. doi:10.1109/SP.2019.00002
[45] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.* IEEE, 75–86.
[46] Jure Leskovec and Rok Sosič. 2016. Snap: A general-purpose network analysis and graph-mining library. *ACM Transactions on Intelligent Systems and Technology (TIST)* 8, 1 (2016), 1–20.
[47] Moritz Lipp, Daniel Gruss, and Michael Schwarz. 2022. {AMD} prefetch attacks through power and time. In *31st USENIX Security Symposium (USENIX Security 22)*. 643–660.
[48] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-Level Cache Side-Channel Attacks are Practical. In *2015 IEEE Symposium on Security and Privacy*. 605–622. doi:10.1109/SP.2015.43
[49] Chi-Keung Luk and Todd C Mowry. 1996. Compiler-based prefetching for recursive data structures. In *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*. 222–233.
[50] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan Berry. 2007. Challenges in parallel graph processing. *Parallel Processing Letters* 17, 01 (2007), 5–20.
[51] Piotr Luszczek, Jack J Dongarra, David Koester, Rolf Rabenseifner, Bob Lucas, Jeremy Kepner, John McCalpin, David Bailey, and Daisuke Takahashi. 2005. Introduction to the HPC challenge benchmark suite. (2005).
[52] Pierre Michaud. 2016. Best-offset hardware prefetching. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 469–480.
[53] Alan Mislove, Massimiliano Marcon, Krishna P Gummadi, Peter Druschel, and Bobby Bhattacharjee. 2007. Measurement and analysis of online social networks. In *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*. 29–42.
[54] Todd C Mowry, Monica S Lam, and Anoop Gupta. 1992. Design and evaluation of a compiler algorithm for prefetching. *ACM Sigplan Notices* 27, 9 (1992), 62–73.
[55] Anurag Mukkara, Nathan Beckmann, Maleen Abeydeera, Xiaosong Ma, and Daniel Sanchez. 2018. Exploiting locality in graph analytics through hardware-accelerated traversal scheduling. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1–14.
[56] Anurag Mukkara, Nathan Beckmann, and Daniel Sanchez. 2019. PHI: Architectural support for synchronization-and bandwidth-efficient commutative scatter updates. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 1009–1022.
[57] Onur Mutlu, Jared Stark, Chris Wilkerson, and Yale N Patt. 2003. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings*. IEEE, 129–140.
[58] Lifeng Nai, Yinglong Xia, Ilie G Tanase, Hyesoon Kim, and Ching-Yung Lin. 2015. GraphBIG: understanding graph computing in the context of industrial solutions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12.
[59] Ajeya Naithani, Sam Ainsworth, Timothy M Jones, and Lieven Eeckhout. 2021. Vector runahead. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 195–208.
[60] Ajeya Naithani, Jaime Roelandts, Sam Ainsworth, Timothy M Jones, and Lieven Eeckhout. 2023. Decoupled vector runahead. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. 17–31.
[61] Agustín Navarro-Torres, Biswabandan Panda, Jesús Alastruey-Benedé, Pablo Ibáñez, Víctor Viñals-Yúfera, and Alberto Ros. 2022. Berti: an accurate local-delta data prefetcher. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 975–991.
[62] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavy-weight dynamic binary instrumentation. *ACM Sigplan notices* 42, 6 (2007), 89–100.

[63] Quan M Nguyen and Daniel Sanchez. 2020. Pipette: Improving core utilization on irregular applications through intra-core pipeline parallelism. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 596–608.

[64] Quan M Nguyen and Daniel Sanchez. 2023. Phloem: Automatic acceleration of irregular applications with fine-grain pipeline parallelism. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 1262–1274.

[65] Intel® oneAPI Programming Guide. 2024. https://www.intel.com/content/www/us/en/developer/tools/oneapi/toolkits.html.

[66] Samuel Pakalapati and Biswabandan Panda. 2020. Bouquet of instruction pointers: Instruction pointer classifier-based spatial hardware prefetching. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 118–131.

[67] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. 2019. Bolt: a practical binary optimizer for data centers and beyond. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2–14.

[68] Leeor Peled, Shie Mannor, Uri Weiser, and Yoav Etsion. 2015. Semantic locality and context-based prefetching using reinforcement learning. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. 285–297.

[69] Shafiur Rahman, Nael Abu-Ghazaleh, and Rajiv Gupta. 2020. Graphpulse: An event-driven hardware accelerator for asynchronous graph processing. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 908–921.

[70] Shafiur Rahman, Mahbod Afarin, Nael Abu-Ghazaleh, and Rajiv Gupta. 2021. Jetstream: Graph analytics on streaming data with event-driven hardware accelerator. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 1091–1105.

[71] Amir Roth and Gurindar S Sohi. 1999. Effective jump-pointer prefetching for linked data structures. In *Proceedings of the 26th annual international symposium on Computer architecture*. 111–121.

[72] David Sehr, Robert Muth, Cliff Biffle, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee, and Brad Chen. 2010. Adapting Software Fault Isolation to Contemporary {CPU} Architectures. In *19th USENIX Security Symposium (USENIX Security 10)*.

[73] Alan Jay Smith. 1978. Sequential program prefetching in memory hierarchies. *Computer* 11, 12 (1978), 7–21.

[74] Bjarne Steensgaard. 1996. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg Beach, Florida, USA) *(POPL '96)*. Association for Computing Machinery, New York, NY, USA, 32–41. doi:10.1145/237721.237727

[75] Nishil Talati, Kyle May, Armand Behroozi, Yichen Yang, Kuba Kaszyk, Christos Vasiladiotis, Tarunesh Verma, Lu Li, Brandon Nguyen, Jiawen Sun, Agreen Ahmadi, Todd Austin, Michael O'Boyle, Scott Mahlke, Trevor Mudge, and Ronald Dreslinski. 2021. Prodigy: Improving the memory latency of data-indirect irregular workloads using hardware-software co-design. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 654–667.

[76] Gang Tan et al. 2017. Principles and implementation techniques of software-based fault isolation. *Foundations and Trends® in Privacy and Security* 1, 3 (2017), 137–198.

[77] Robert Wahbe, Steven Lucco, Thomas E Anderson, and Susan L Graham. 1993. Efficient software-based fault isolation. In *Proceedings of the fourteenth ACM symposium on Operating systems principles*. 203–216.

[78] Hao Wu, Krishnendra Nathella, Joseph Pusdesris, Dam Sunwoo, Akanksha Jain, and Calvin Lin. 2019. Temporal prefetching without the off-chip metadata. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 996–1008.

[79] Hao Wu, Krishnendra Nathella, Dam Sunwoo, Akanksha Jain, and Calvin Lin. 2019. Efficient metadata management for irregular data prefetching. In *Proceedings of the 46th International Symposium on Computer Architecture*. 449–461.

[80] Youfeng Wu. 2002. Efficient discovery of regular stride patterns in irregular programs and its use in compiler prefetching. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*. 210–221.

[81] Tian Xia, Gelin Fu, Chenyang Li, Zhongpei Luo, Lucheng Zhang, Ruiyang Chen, Wenzhe Zhao, Nanning Zheng, and Pengju Ren. 2022. A comprehensive performance model of sparse matrix-vector multiplication to guide kernel optimization. *IEEE Transactions on Parallel and Distributed Systems* 34, 2 (2022), 519–534.

[82] Feng Xue, Chenji Han, Xinyu Li, Junliang Wu, Tingting Zhang, Tianyi Liu, Yifan Hao, Zidong Du, Qi Guo, and Fuxin Zhang. 2024. Tyche: An Efficient and General Prefetcher for Indirect Memory Accesses. *ACM Transactions on Architecture and Code Optimization* 21, 2 (2024), 1–26.

[83] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *23rd USENIX Security Symposium (USENIX Security 14)*. USENIX Association, San Diego, CA, 719–732. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom

[84] Xiangyao Yu, Christopher J Hughes, Nadathur Satish, and Srinivas Devadas. 2015. IMP: Indirect memory prefetcher. In *Proceedings of the 48th International Symposium on Microarchitecture*. 178–190.

[85] Yuxuan Zhang, Nathan Sobotka, Soyoon Park, Saba Jamilan, Tanvir Ahmed Khan, Baris Kasikci, Gilles A Pokam, Heiner Litz, and Joseph Devietti. 2024. RPG2: Robust Profile-Guided Runtime Prefetch Generation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 999–1013.